ALMA MATER STUDIORUM - UNIVERSITÀ
DI BOLOGNA CESENA CAMPUS

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING – DISI

SECOND CYCLE DEGREE IN
DIGITAL TRANSFORMATION MANAGEMENT

Class: LM-91

The Evolution of Coding in the Digital Transformation Era Cybersecurity Implications of Artificial Intelligence and Low-Code Development

Graduation thesis in
Cybersecurity

Supervisor
Gabriele D'Angelo

Co-Supervisor
Gian Luigi Bonini

Candidate
Domenico Copia

Academic Year 2024/25

Session III

Abstract

The accelerating wave of digital transformation has made software the core infrastructure of modern competitiveness. Yet, as organizations adopt artificial intelligence (AI) and low-code/no-code tools to accelerate development, they also expose new vectors of vulnerability, opacity and technical debt. This thesis examines how the evolution of software creation, from traditional programming to AI-assisted and low-code paradigms, reshapes the cybersecurity and governance landscape.

The research addresses three key questions: (1) how AI-assisted and low-code development alter software vulnerabilities and accountability; (2) which governance and assurance mechanisms can mitigate emerging risks without undermining agility; and (3) how standardized frameworks can institutionalize quality and security in increasingly automated environments. Methodologically, the study combines literature review, risk analysis, and framework design, supported by the Italian initiative Innovation Code as a reference for certification and secure-by-design principles.

Findings show that AI-generated and low-code software enhance productivity and democratize innovation but also erode transparency, expand attack surfaces, and increase reliance on opaque models. Addressing these issues requires integrating technical safeguards, automated testing, software bills of materials, provenance tracking, with governance measures such as certification, standardization, and regulatory alignment under NIS2 and the EU AI Act.

The thesis contributes a conceptual and operational framework for secure, compliant, and sustainable software development in the AI era, concluding that the future of coding lies not in replacing human expertise but in orchestrating intelligent tools and certified components within transparent, auditable, and collaborative ecosystems where speed and security coexist in balance.

Contents

Introduction

In the contemporary landscape, digital transformation has become the primary engine of economic, social, and organizational change. Across industries, the progressive integration of software, data, and intelligent automation has reshaped the boundaries between technology and management, redefining how value is created, delivered, and protected. Within this scenario, software is no longer a mere operational support but the core infrastructure of competitiveness. Its strategic role, however, is paralleled by increasing exposure to vulnerabilities, dependencies, and governance challenges.

This thesis investigates the intersection between **software evolution**, **cybersecurity**, and **digital transformation**, with particular attention to the impact of **artificial intelligence (AI)** and **low-code/no-code development** on software quality and security. The research originates from a growing concern: as organizations strive to accelerate delivery and innovation, the complexity and opacity of software systems expand faster than their capacity to ensure control and assurance. Large language models now assist developers in generating code, while low-code platforms allow non-technical users to build applications with minimal programming effort. These innovations democratize software creation but also blur accountability, amplify technical debt, and expose new classes of systemic risks.

The thesis adopts a dual perspective, **technological and managerial.** On one hand, it explores the technical mechanisms that underlie AI-assisted and low-code development, identifying their efficiency drivers and vulnerability points. On the other, it examines how governance frameworks, standards, and regulatory initiatives can bridge the gap between speed and security. The study aligns with the broader European discourse on **software trustworthiness**, as embodied in frameworks such as the EU Cybersecurity Act, the NIS2 Directive, and the recently adopted AI Act, which together emphasize transparency, auditability, and shared responsibility in digital innovation.

Within this context, the Italian initiative **“Innovation Code”** plays a central role in the research. Conceived as a national framework for software quality and certification, it aims to create a standardized, reusable, and secure digital ecosystem through certified components, automation, and AI governance. The project offers a tangible reference point for analyzing how secure-by-design principles and regulatory compliance can coexist with rapid, AI-enabled development practices.

The ultimate purpose of this thesis is to contribute to the understanding of how **software engineering is evolving** in the age of automation and regulation. It seeks to define strategies for minimizing technical debt, improving security assurance, and aligning innovation with compliance. In doing so, it argues that the future of software creation will depend less on manual coding and more on orchestrating intelligent tools, certified components, and standardized processes. The challenge and opportunity lies in ensuring that this acceleration remains sustainable, ethical, and secure.

Through a multidisciplinary approach that combines insights from computer science, cybersecurity, and management, this research aspires to provide both theoretical understanding and actionable guidance for organizations, policymakers, and professionals navigating the new frontiers of digital transformation.

Chapter 1

Software, Security and Digital Transformation

The digital transformation of the past two decades has reshaped the foundations of how societies and organizations create value. What began as a process of digitization, converting analog processes into digital form, has evolved into a profound structural transformation that touches every aspect of production, communication, and decision-making. In this new paradigm, technological change is not an external enabler but an intrinsic component of business strategy and governance. Software, data, and artificial intelligence no longer serve as auxiliary tools; they have become the infrastructure upon which economic and social systems operate.

This shift has altered the meaning of competitiveness and innovation. The capacity to generate insights from data, automate complex tasks, and coordinate global operations in real time has created new forms of advantage, while also introducing new dependencies and risks. Every organization now functions as a software organization to some degree, relying on interconnected platforms, algorithms, and digital ecosystems that transcend traditional boundaries. As a result, technological interdependence has deepened, creating both extraordinary potential and systemic fragility.

Digital transformation thus represents a dual movement: one of empowerment and one of exposure. The same networks and applications that enable speed and intelligence also expand the space for failure, misuse or attack. The increasing integration of artificial intelligence and low-code development further amplifies this tension, accelerating innovation while complicating oversight and control. Understanding this paradox is essential for grasping the current state of the digital economy and the challenges of governing its evolution.

The following discussion unfolds within this broader context, examining how digital transformation has moved from infrastructure to intelligence, how software has become the central asset of the modern enterprise, and how complexity has turned innovation itself into a source of vulnerability. This sets the conceptual ground for exploring the emerging need for new governance models that reconcile agility with security in the age of intelligent software systems.

1.1 The Digital Transformation Era: From Infrastructure to Intelligence

Digital transformation can be understood as a systemic shift encompassing organizational, technological, and cultural changes in how value is created. It is often described as the fundamental *rewiring* of how an organization operates, “with the goal of building competitive advantage by continuously deploying technology at scale” [1]. This shift is not a one-off IT project but a long-term evolution that demands new strategies, talent, and mindsets across the enterprise. Crucially, digital transformation goes beyond merely digitizing existing processes, it calls for rethinking business models and workflows around data and interconnectivity, thereby integrating technology deeply into organizational DNA.

One hallmark of the digital era is the move from competition based on physical assets and infrastructure to competition based on data and software-driven capabilities. In traditional industrial economies, capital investments and tangible assets (factories, machinery, real estate) largely determined a firm’s competitive edge. Today, however, intangible digital assets, such as software platforms, algorithms, and proprietary data, have become primary sources of value creation. Organizations increasingly rely on software to streamline operations and innovate, gaining competitive advantages through superior data analytics and automation. For instance, companies that rapidly adopt new digital technologies can attain significant competitive advantages over rivals, positioning themselves favorably in the market [2]. Empirical trends underscore this shift: firms primarily valued for their software offerings have grown dramatically in market significance (expanding from about 2% to 13% of market share between 1996 and 2023) [3]. In short, **the basis of competitive success has expanded** from owning infrastructure to harnessing intelligence, the ability to collect, process and **act on information faster and more effectively than the competition**.

Another defining feature of this era is the rise of Artificial Intelligence (AI) as the new “cognitive infrastructure” of digital enterprises. Just as electricity and the internet served as general-purpose infrastructures in previous eras, AI now functions as a **foundational layer** enabling advanced capabilities across business domains. Rather than being viewed as a standalone product or just another application, AI is increasingly seen as an embedded intelligence that permeates organizational processes. Modern AI systems, from machine learning models to cognitive services, provide a decision-support and pattern-recognition infrastructure that augments human intelligence in areas like forecasting, customer service, and strategic planning. In practical terms, AI-driven analytics and automation act as a “thinking infrastructure” for the enterprise, enhancing decision-making without wholly replacing human judgment [4]. This convergence of AI with core business functions marks a shift from an era focused on digitizing infrastructure to an era focused on intelligent infrastructure. In summary, the Digital Transformation Era is characterized by a transition from traditional infrastructure-based value creation to intelligence-based value creation, where *data*, *software*, and *AI* form the nexus of competitive advantage and organizational innovation.

1.2 Software as the Core Asset of the Digital Economy

Software has evolved from a supportive operational tool into a central strategic asset in the digital economy. In earlier decades, software primarily served to automate back-office tasks or enable efficiency in defined processes; today it is integral to virtually every facet of business value delivery. Software technologies are now deeply embedded in nearly every industry and daily activity, from healthcare and transportation to finance and media. Organizations no longer see software as peripheral infrastructure, but as *strategic capital*, a source of innovation and differentiation in its own right. Notably, enterprises leverage software not just to support operations, but to drive new business models (for example, software-as-a-service and platform-based ecosystems) and to inform high-level decision-making through analytics. As a result, the ability to develop, acquire, and maintain high-quality software has become synonymous with the ability to compete and adapt in the modern economy. Indeed, companies that excel in software development and integration tend to outperform peers, as they can streamline workflows and rapidly deploy new services or products. The close interdependence of software with decision-making, automation, and system scalability means that business growth and agility increasingly hinge on software capabilities. Many organizational decisions (from daily operational choices to strategic planning) are now guided by software-driven insights such as data analytics and AI algorithms. Likewise, the scalability of a modern enterprise, its capacity to serve millions of users or process big data in real time, relies on robust software architectures in the cloud and on networks. In essence, software has become the *brain and nervous system* of contemporary organizations.

This centrality of software brings tremendous benefits, but it also introduces a new form of systemic vulnerability: *software dependence*. As firms and economies become ever more dependent on complex software systems, any fragility or flaw in those systems can have far-reaching consequences. A bug in widely-used software or a failure in a critical application, can disrupt not only a single company's operations but also reverberate across supply chains and customer networks. Put differently, software has become a single point of failure on a systemic scale, a reality highlighted by recent incidents. For example, the "Heartbleed" bug in the OpenSSL cryptographic library (discovered in 2014) [5] exposed a vast number of web servers to potential data breach, demonstrating how a vulnerability in a common software component can jeopardize security globally. In short, software's central role in the digital economy creates a paradox: the same interconnectedness and reuse that enable innovation also increase the risk of widespread failures or attacks. For this reason, organizations must view software not only as a strategic asset but also as a potential source of systemic risk, making security and quality assurance essential parts of every development and deployment process.

1.3 The Security Paradox: Innovation, Complexity, and Exposure

Rapid innovation and increasing automation have yielded extraordinary capabilities, but they have also inadvertently increased system complexity and reduced direct human oversight. As organizations digitalize, they integrate numerous technologies, cloud platforms, IoT devices, AI services, APIs and more, into intricate architectures. Each new integration or automated process adds layers of complexity that can outstrip a single individual's or team's ability to fully understand or control. The result is often an *opacity* in how systems operate: in highly automated environments, humans become overseers of complex autonomous processes rather than hands-on controllers of each function. This specialization and complexity create a *delusion of control*, where systems run with minimal human intervention until an unexpected condition arises that requires manual correction. By that time, the system's behavior may be so complex that diagnosing and fixing issues is challenging. In essence, innovation has traded direct human control for algorithmic or procedural control, which can fail in unpredictable ways.

One consequence of this complexity is a dramatic expansion of the potential attack surface and an emergence of new vulnerabilities. The *attack surface* refers to all possible entry points that an attacker could exploit to gain unauthorized access to a system. **Complexity is the enemy of security**: as more components and connections exist, there are more points where things can go wrong or be exploited [6]. Modern digital systems comprise a multitude of software modules, microservices, and third-party APIs; identifying every vulnerability among them, let alone patching all in a timely manner, becomes a daunting task [6]. Importantly, several categories of emerging technology bring their own security paradoxes:

Artificial Intelligence (AI): AI systems introduce new types of errors and attack methods. Machine learning models can be deceived by *adversarial inputs* (maliciously crafted data that causes the model to make wrong decisions), while AI-based code generation tools may produce insecure code if not properly controlled. Experiments have shown that almost half of the code generated by popular AI assistants contained bugs or vulnerabilities that could be exploited [7]. In addition, AI models themselves can be targeted through *data poisoning* (tampering with the training data) or *model theft* (copying or extracting the trained model), creating a new category of risks specific to systems that learn and evolve over time.

APIs and Interconnectivity: The growing use of web services and APIs means that most modern applications communicate through public interfaces. An *API* (Application Programming Interface) is a set of rules that allows different software systems to exchange data and functions, for example, when a mobile app retrieves information from a cloud service. While this modular design promotes flexibility and integration, it also exposes many functionalities to the internet. Because APIs now handle about 71% of all web traffic [8], they have become major targets for attackers. Each API endpoint is a potential entry point, and if it is misconfigured or lacks proper security controls, it can lead to serious data breaches. Managing dozens or even hundreds of APIs across cloud and on-premise systems increases the risk that some remain unprotected or forgotten.

Low-Code/No-Code Platforms: Low-code tools allow non-experts to build applications quickly, but this accessibility introduces security trade-offs. Because these apps often skip the thorough reviews and testing used in traditional development, they may contain hidden risks. In this context, *citizen developers* are employees or users without formal programming training who create software using visual, drag-and-drop tools instead of writing code. While this democratization of development speeds up innovation, it can also lead to misconfigurations, such as granting incorrect permissions or exposing sensitive data, if proper governance is not in place. Therefore, low-code platforms must always be accompanied by clear security guidance and supervision.

Software Supply Chain: Modern software is built from many external and open-source components. This dependence means that a single flaw in a popular library can expose thousands of organizations at once. Because these components are interconnected, one vulnerability can spread through the entire ecosystem like a chain reaction. These are known as *software supply chain attacks*, where attackers compromise weaker links (for example, a small vendor or open-source maintainer) to reach the larger systems that depend on them.

In light of these challenges, there is a growing recognition that **cybersecurity must be reframed as an enabler of innovation, not a barrier**. In the past, security was sometimes viewed as a drag on agility, strict controls and checks that slowed down release cycles or constrained new features. However, as digital transformation has made business success inseparable from technology, robust cybersecurity is increasingly seen as foundational to sustained innovation. Strong security postures create the trust and stability required to experiment with new digital offerings. Industry leaders now assert that cybersecurity is not just a safeguard but a strategic enabler of innovation, customer trust, and long-term growth [9]. By embedding security into the design of systems (secure-by-design principles) and into governance processes, organizations can pursue rapid development *and* remain resilient to threats. In practice, this means integrating security teams directly into development and operations, an approach known as *DevSecOps*, where security is built into every stage of the software lifecycle rather than added at the end. It also involves adopting frameworks that simplify and organize security architectures to reduce complexity, and ensuring that cybersecurity investments align with business goals and governance priorities. Notably, modern regulations and standards encourage this alignment: for example, the EU's NIS2 Directive explicitly elevates cybersecurity to a board-level responsibility, ensuring that innovation initiatives are paired with accountability for managing cyber risks [10]. The security paradox can thus be resolved by treating cybersecurity not as a hindrance, but as a critical success factor, one that enables organizations to innovate *safely* in an environment of complex, distributed digital systems.

1.4 Research Motivation and Objectives

The core motivation for this research is to explore the intersection of cutting-edge software development paradigms and cybersecurity. With the advent of AI-assisted coding tools and low-code/no-code development platforms, software creation is becoming faster and more accessible, but potentially at the expense of quality and security. This thesis is driven by the need to understand the *security and quality implications* of these trends. In particular, the work seeks to illuminate how the ongoing digital transformation reshapes software's role and risk, what new vulnerabilities are emerging from AI and low-code practices, and how organizations might govern technology in a way that balances rapid innovation with robust security.

To address this motivation, the study is organized around three central research questions:

1. **How do AI-assisted development and low-code platforms alter the nature of software vulnerabilities and accountability?**
   This question explores how emerging development paradigms shift the technical and organizational boundaries of responsibility. By automating parts of the coding process or enabling non-expert users to build applications, these tools redefine who is accountable for code quality and security. The inquiry examines empirical evidence showing that AI-generated code may replicate insecure patterns, while low-code platforms can introduce misconfigurations and hidden dependencies when used without proper governance.

2. **Which governance and assurance mechanisms can mitigate emerging risks without undermining agility?**
   Here, the focus turns to management and control frameworks that reconcile speed and safety. It investigates how organizations can integrate automated checks, quality certification, and continuous security assurance into rapid development cycles, ensuring compliance with regulations such as NIS2 [10] and the EU AI Act while preserving flexibility and innovation.

3. **How can standardized frameworks institutionalize quality and security in increasingly automated environments?**
   The third question broadens the analysis to a systemic level. It examines how initiatives such as *Innovation Code* and global standards (e.g. ISO/IEC 25010, SLSA, SBOM) can serve as anchors for sustainable digital transformation. The goal is to understand whether shared governance and standardized certification can embed cybersecurity and quality management into the fabric of innovation itself, transforming security from a reactive safeguard into a proactive enabler.

Together, these questions frame the thesis around a fundamental challenge: designing a software ecosystem where automation, intelligence, and regulation coexist in balance, enabling innovation that is not only fast, but also accountable, auditable, and secure by design.

The objective of this thesis, corresponding to these questions, is to conduct a **critical analysis of current trends and frameworks** and to lay the groundwork for a practical initiative addressing these issues. Concretely, the study aims to:

1. Analyze how digital transformation has elevated the role of software and the nature of software-related risks (drawing on literature and industry observations).

2. Identify and examine vulnerabilities emerging from AI-assisted coding and low-code development, comparing them to traditional software vulnerabilities.
3. Investigate whether and how innovation can coevolve with security through collaborative governance, by comparing frameworks and standards (such as OWASP best practices, the European NIS2 directive, and the ISO/IEC 25010 software quality model).
4. Finally, as a forward-looking component, preview the **“Innovation Code”** initiative, an industry-driven program, evaluating it as a case that embodies these themes and potentially offers solutions.

By meeting these objectives, the thesis will bridge theoretical and practical perspectives. It will not only survey and synthesize existing knowledge (on digital transformation, AI risks, governance models, etc.) but also use that understanding to assess new approaches (like Innovation Code) that aim to reconcile the demands of innovation speed, quality, and security. The overall goal is to derive insights and recommendations that can guide both practitioners (in managing real-world software projects) and policymakers or industry groups working on frameworks for secure digital innovation.

1.5 From Problem to Solution: The Genesis of the “Innovation Code” Initiative

In response to the challenges outlined above, ranging from the increasing complexity of digital systems to the difficulty of maintaining both speed and control in software development, the Italian initiative “Innovation Code” has emerged as a coordinated, forward-looking response. Conceived within the industrial and technological ecosystem of Italy, and particularly promoted by Confindustria Romagna, the initiative was born out of a shared awareness among companies, ICT and professionals, that the traditional, fragmented approaches to software governance were no longer sufficient. Rather than relying on isolated best practices or reactive compliance measures, Innovation Code proposes a systemic model designed to embed quality, security, and sustainability into the entire software lifecycle.

The initiative is structured around four key pillars: quality, reuse, governance, and sustainability. It recognizes that many of the weaknesses found in modern software, ranging from technical debt to inconsistent coding standards, stem not from individual mistakes but from the absence of shared frameworks and collective accountability. To address this, Innovation Code promotes the creation of a standardized environment where organizations can develop and share certified software components, reuse proven modules, and operate within a transparent and auditable governance model. This shared repository of trusted components aims to reduce redundant effort, minimize vulnerabilities, and ensure that updates or patches can be deployed consistently across projects, reinforcing security and efficiency at scale.

Confindustria Romagna, as the driving institutional partner, has played a pivotal role in fostering collaboration between industry and academia, connecting enterprises of varying sizes, especially SMEs, with innovation partners and digital experts. Through this networked approach, Innovation Code functions not merely as a technical framework but as a strategic ecosystem that encourages responsible innovation. It brings together coordinators, authors, and users in a tiered participation model, ensuring that expertise, oversight, and operational execution are balanced. This collaborative structure ensures that rapid innovation through AI-assisted and low-code tools can coexist with rigorous quality and cybersecurity governance.

Although the specific mechanisms, certifications, and pilot projects of Innovation Code will be examined later, its introduction here serves a wider purpose. The initiative represents a clear example of how modern software ecosystems can balance two essential goals: moving fast while remaining accountable. In this sense, Innovation Code reflects a governance approach that aligns with European priorities set out in the NIS2 Directive and the AI Act, both of which emphasize transparency, traceability, and shared responsibility as the foundation of trustworthy digital innovation.

This initiative provides both the conceptual and practical foundation for the main question guiding this thesis: how can innovation and cybersecurity grow together under structured governance? It acts as a bridge between theory and practice, showing how organizations can apply secure-by-design principles without slowing down technological progress.

The transition to Chapter 2 follows naturally from this discussion. After exploring governance models and systemic responses such as Innovation Code, the next chapter shifts focus to the technological side of transformation. It will trace the evolution of coding itself, from traditional programming to AI-assisted and low-code approaches, highlighting how these innovations have reshaped not only the way software is developed but also how its security and reliability must be reconsidered in today’s digital era.

Chapter 2

The Evolution of Coding: Between AI, Low-Code and Emerging Languages

The art and science of programming have undergone profound transformations since the early days of computing, evolving from arcane sequences of machine code to sophisticated systems capable of interpreting natural language. Each generational leap in abstraction has not only changed how software is written but also who is empowered to write it. This chapter explores the major technological cycles that have shaped software development, tracing a historical arc from low-level coding practices to the emergence of artificial intelligence, low-code/no-code platforms and natural language programming. These shifts are not merely technical, they carry far-reaching implications for software security, governance and the role of the human developer.

The driving force behind this evolution has consistently been the pursuit of greater productivity, accessibility and expressiveness. High-level compiled languages, object-oriented paradigms and fourth-generation languages each played their part in abstracting away hardware constraints and democratizing software creation. More recently, the irruption of AI into development environments, exemplified by tools like GitHub Copilot, has begun to redefine coding itself, transforming natural language into a de facto programming interface. Parallel to this, low-code and no-code platforms are enabling a new generation of "citizen developers" to participate in application development, often without formal programming backgrounds.

These trends, while empowering, introduce new challenges for software quality and security. The delegation of code generation to AI or non-technical users raises critical questions about the integrity, safety and maintainability of modern software systems. As abstraction increases, so too does the risk of hidden vulnerabilities and the need for robust oversight mechanisms. This chapter examines these dynamics in depth, offering both a historical perspective and a forward-looking analysis of where software development is headed and what it means for the secure design of tomorrow's systems.

In doing so, it sets the foundation for understanding how emerging coding paradigms (from AI-generated code to fully declarative natural language interfaces) are reshaping the landscape of cybersecurity and what strategies will be necessary to navigate this new era responsibly.

2.1 A Brief History and Technological Cycles of Coding

The practice of coding has undergone several transformative cycles since the dawn of computing. In the earliest era (1940s-1950s), programming meant writing low-level instructions in machine code or assembly language, painstakingly toggling switches or punching cards. The late 1950s and 1960s introduced high-level *compiled* languages like Fortran and COBOL, which abstracted away hardware details and greatly increased developer productivity. Each leap in abstraction (from machine code to compiled languages and later to interpreted and managed languages) broadened accessibility to programming. Moving from assembly to compiled code meant that a programmer "didn't have to understand what registers are… you just needed to understand if-then-else or what a variable assignment is". [11] Similarly, the transition from compiled languages (like C/C++) to higher-level interpreted languages and frameworks (like Java, JavaScript, Python in the 1990s) further lowered the entry barriers and enabled more rapid development. Each cycle of abstraction initially met resistance from traditionalists who feared a "loss of knowledge", yet time and again these fears proved unfounded as productivity gains were realized.

By the 1980s, new programming paradigms were emerging, including object-oriented programming (e.g. C++ and later Java) and fourth-generation languages (4GLs). 4GLs marked an important shift: instead of requiring developers to write detailed step-by-step instructions (as with earlier languages like C), they allowed them to simply specify what the program should achieve. The system would then figure out how to perform the task. This approach anticipated today's declarative and visual development tools.

At the same time, **Computer-Aided Software Engineering (CASE)** tools began to appear. These were specialized programs designed to automate parts of the software development process, such as designing database structures, generating code or managing documentation. In essence, CASE tools aimed to make coding faster, more consistent and more accessible by providing visual editors, templates and automatic code generation features.

Visionaries like James Martin predicted, as early as 1982 in *Application Development Without Programmers*, that the shortage of skilled developers would drive the need for tools that could enable non-programmers to build applications. Indeed, during the 1980s, CASE and 4GL platforms promised to democratize software creation by minimizing the amount of manual coding required. These early technologies were the forerunners of what we now call **citizen development platforms**, tools that empower non-technical users to create software solutions [12]. (This concept of the "citizen developer" will be explored more deeply later in the chapter.)

However, despite the enthusiasm, these early attempts often struggled in practice: many CASE and 4GL systems failed to scale effectively for complex applications and often lacked essential capabilities like version control and thorough testing frameworks. As a result, by the late 1990s, attention shifted toward new programming languages (such as JavaScript and PHP) and frameworks driven by the rise of the web and client-server architectures, temporarily sidelining those early “low-code” ideas.

Timeline of Key Disruptions

To illustrate the cycles, consider a brief timeline. In **1957**, the introduction of FORTRAN (the first widely used high-level language) was a breakthrough that simplified numeric computing. The **1970s** brought C and structured programming, enabling development of complex systems like operating systems with more manageable code. The **1980s** saw the advent of PC software and object-oriented languages (C++, Smalltalk) and also early visual programming experiments (e.g. Visual Basic emerged in 1991 blending GUI design with coding). The **1990s** and **2000s** were dominated by the internet boom, scripting languages (Perl, Python, JavaScript) and enterprise frameworks (Java EE, .NET) a shift towards rapid development and reuse of libraries. Each of these phases demonstrates a cycle of rising abstraction: from hardware-near coding towards higher-level, human-friendly constructs. Now, in the late **2010s** and **2020s**, we are in the midst of another quantum leap, one where **artificial intelligence** and **low-code/no-code platforms** radically redefine how software is created. The following sections examine this modern evolution and its implications, particularly for security.

2.2 The Irruption of Artificial Intelligence in Software Development

Recent years have seen an unprecedented **irruption of Artificial Intelligence (AI)** into the realm of software development. AI-powered tools are now writing code, debugging and even architecting software alongside human developers. The launch of GitHub Copilot in 2021, described as an "AI pair programmer", was a watershed moment. Copilot and similar generative AI models (Anthropic Claude, OpenAI ChatGPT, Amazon CodeWhisperer, etc.) use large language models trained on vast code corpora to suggest code completions or even generate entire functions based on natural-language prompts. This represents a new paradigm: instead of manually writing every line, a developer can now **describe the intended functionality in English** (or any other natural language) and let the AI produce candidate code. As Nvidia's CEO Jensen Huang put it, *"the programming language is human"* and in the future *"you will tell the computer what you want and it will do it"* [16]. In other words, AI is turning English (or any human language) into a universal programming language by serving as the translator from intent to implementation.

The integration of AI has already shown tangible productivity benefits. Studies found that developers using AI assistants can complete tasks significantly faster. For example, an experiment by GitHub revealed that programmers with Copilot finished a coding task **55% faster** on average than those without it [18]. This speed-up (reducing a task that took 2 hours 41 minutes down to 1 hour 11 minutes in the study) underscores AI's potential to automate boilerplate coding and accelerate development. Beyond speed, AI coding assistants can improve developer experience: surveys report that a majority of developers feel less frustrated and more "in the flow" when using AI assistance, allowing them to focus on creative and complex aspects of software design. Essentially, AI can handle repetitive grunt work (e.g. writing routine functions, suggesting syntax) so that human developers can concentrate on **higher-level problem solving**.

However, the irruption of AI also brings significant **cybersecurity implications and challenges**. One concern is the **security of AI-generated code**. Since these models learn from public code (which may include insecure patterns), they can inadvertently produce vulnerable code constructs. A prominent study in 2022 systematically analyzed Copilot's outputs for secure coding practices and found that roughly **40% of the AI-generated programs had security vulnerabilities** [17]. In scenarios targeting common weakness patterns (such as those in MITRE's CWE Top 25, detailed in the next chapter), Copilot often suggested solutions that were functionally correct but insecure, for example using outdated encryption or susceptible SQL queries. These findings highlight the risk that AI assistance might introduce hidden flaws if a human developer accepts suggestions without scrutiny. In response, tool makers have started adding AI-based vulnerability filters to block obviously insecure suggestions. Nevertheless, **the onus remains on developers** to review and test AI-written code carefully, effectively shifting their role from writing code to **auditing and curating code.** This shift itself is a major change: developers must develop new skills in prompt engineering (to get useful outputs from the AI) and in security-aware code review, as they guide and correct their AI collaborators.

Another challenge is the potential for **malicious use of AI in coding**. Just as AI can help developers build apps faster, it could assist attackers in generating malware or exploits. There is concern that generative models could lower the barrier for creating sophisticated attacks by auto-coding malicious scripts or polymorphic code. On the flip side, AI is also being harnessed for defensive security e.g. AI systems that refactor insecure code or generate fixes for known vulnerabilities. This dual-edged nature of AI in development means cybersecurity professionals must stay vigilant: the development landscape is evolving where AI can be co-developer or adversary. In summary, AI's sudden entrance into software engineering is transforming how code is written. It promises unprecedented automation and efficiency, the ability to generate code from natural language is "another rung up the ladder of abstraction" beyond even high-level languages. Yet it also requires a rethinking of secure development life cycles, with new practices to ensure AI-generated code does not become the weakest link in security.

The next section will explore a parallel revolution making coding more accessible: the rise of low-code and no-code development, which intersects with the AI trend to further democratize software creation.

2.3 The Low-Code/No-Code Paradigm and the Rise of the "Citizen Developer"

In tandem with AI, the software industry is experiencing a renaissance of **low-code/no-code development platforms**. These platforms allow applications to be built with minimal hand-written code, often through graphical interfaces, drag-and-drop components and declarative configuration. The concept is not entirely new as noted, the roots trace back to 4GLs in the late 20th century, but modern low-code platforms are far more capable and integrated. Their resurgence is driven by a critical need: businesses face a **shortage of professional developers** and a backlog of software needs, so empowering non-programmers to create software is an attractive solution. The term *"citizen developer"* has emerged to describe business users (outside the IT department) who build or customize applications using IT-sanctioned low-code/no-code tools, rather than traditional coding [13]. This paradigm shift enables domain experts e.g. a finance analyst automating a reporting workflow, to become creators of software solutions without deep programming skills.

Modern low-code/no-code platforms include examples like **WaveMaker, Oracle APEX** and **Microsoft Power Apps**, among many others. These tools exemplify the low-code approach in practice. *WaveMaker* is a Java-based low-code platform oriented toward professional developers building enterprise apps; it provides a visual studio and component library on top of an open standards stack (Java/Spring, etc.), allowing developers to "leverage the speed of low-code with control over custom coding" [14]. *Oracle APEX* (Application Express) is another platform, originally developed by Oracle in the 2000s, that enables rapid development of web applications with a browser-based interface. APEX exemplifies a **hybrid development model**: one can assemble an app with wizards and drag-drop UI designers, achieving a functional prototype without writing code, but if needed, developers can extend or fine-tune the app with SQL and PL/SQL logic or by injecting JavaScript for custom behaviors [15]. *Microsoft Power Apps* similarly allows creation of business applications through a visual canvas and Excel-like formulas, tightly integrating with the Office 365 ecosystem for citizen developers in organizations.

As introduced in Chapter 1, in line with the growing adoption of low-code solutions across industries and governments, the **Innovation Code** project stands out as a pioneering initiative driving digital transformation within the Italian ecosystem. Launched by **Meta**, the innovation arm of **Confindustria Romagna**, Innovation Code was created to address the digital gap faced by small and medium-sized enterprises by promoting the use of low-code technologies such as the ones just discussed.

Innovation Code operates as a structured community, bringing together companies, developers and IT professionals **under a shared framework of collaboration**. Participants are organized into different levels: Coordinators, Authors and Users. each with specific responsibilities and rights. Central to the initiative is a **curated marketplace**, where certified software components, modules and APIs are made available. This enables businesses to quickly assemble customized solutions while significantly reducing development time and cost. The initiative's structure, certification process, and governance model offer much to explore and will be examined in greater depth in Chapter 5.

Innovation Code is a prime example of how low-code tools can bridge the software demand gap by empowering a wider range of contributors, including those with limited technical backgrounds, a concept that will be further explored when discussing the rise of **citizen development** later in this chapter.

The **advantages** of low-code/no-code are clear: faster development cycles, lower barrier to create simple apps and the ability to involve end-users directly in building the tools they need. Gartner predicts that by 2026, **developers outside formal IT departments will account for at least 80% of the user base for low-code development tools**, a dramatic increase from just a few years prior [20]. These numbers underscore that citizen development is moving into the mainstream. From a business perspective, this helps alleviate the developer talent shortage and allows power users to "solve their own problems" directly. When done right, it can boost innovation and efficiency, freeing IT staff to focus on more complex, mission-critical projects while line-of-business teams handle lighter applications.

Security and governance concerns: Despite its promise, the low-code movement brings challenges, particularly for cybersecurity and IT governance. Allowing a wide swath of non-engineers to create applications can potentially **amplify security risks** if not managed properly. Earlier generations of pseudo-coding tools learned this the hard way: many early 4GL projects in the '80s and '90s resulted in fragile, unsecure applications. As one analysis noted, empowering non-technical people to build software "exposed the organization to several risks, chief among them that most non-technical builders did not possess the skillset to create and deploy applications with appropriate security and governance" [12]. In the modern era, low-code platforms have improved on this by baking in security features and offering admin oversight. Most enterprise-grade low-code platforms now include centralized IT governance, role-based access control and compliance certifications. For example, platforms delivered via the cloud can enforce updates and security patches universally. Additionally, the maturation of the user base and best practices in the last decade have made it easier to establish guidelines for citizen developers (e.g. requiring IT review of apps that use sensitive data). Nonetheless, **organizations must implement proper governance**: establishing which data and systems citizen-developed apps can access, providing training on secure practices for citizen devs and monitoring for compliance. Without these measures, low-code apps could inadvertently become a new attack surface (through misconfigured data access, lack of encryption, etc.).

Another concern is **scalability and maintainability** of low-code solutions. If dozens of departments build their own mini-apps, an IT department could face a sprawl of semi-supported tools. This is why many companies create "fusion teams" pairing citizen developers with professional developers to ensure that the resulting software meets quality standards and can be maintained or integrated properly long-term. The "citizen developer" trend thus forces a redefinition of roles: the pro developers take on more of a curator/mentor role, setting up the guardrails and stepping in to extend platforms when custom code is necessary, while the business users contribute domain knowledge and quick prototyping.

From a strategic perspective, low-code and no-code platforms are democratizing software development by extending participation beyond traditional IT roles. However, this democratization must be managed carefully to avoid introducing significant risks to cybersecurity and governance. Rather than embracing an uncontrolled expansion of "citizen developers", initiatives such as **Innovation Code** advocate for a more disciplined model, contrary to the simplistic notion that "everyone can build their own apps" a more sustainable and secure approach involves: selecting individuals with STEM backgrounds or strong technical aptitudes and providing them with structured training in development standards, security principles and platform governance.

This model preserves the advantages of low-code, such as faster development cycles and reduced pressure on IT departments while minimizing the risks typically associated with fragmented or ad hoc software creation. By embedding governance mechanisms directly into the development process and ensuring that new contributors are appropriately trained, organizations can expand their innovation capacity without compromising security or operational integrity. As low-code adoption accelerates, striking a balance between empowering a broader workforce and maintaining rigorous oversight will be essential. The next frontier will push these boundaries even further, with the emergence of hybrid models and natural language programming, topics explored in the following section.

2.4 New Frontiers of Coding: Hybrid Languages and Natural Language Programming

The evolution of coding paradigms is now reaching a point where the lines between human language and programming language begin to blur. Two notable frontiers are emerging: **hybrid languages** and **natural language programming**. These developments aim to make programming more expressive, intuitive and aligned with human thinking, which could fundamentally change how we approach software development.

Hybrid Languages: The term "hybrid languages" refers to programming languages or environments that integrate multiple programming paradigms, that is, distinct styles or models of organizing and thinking about software development. A programming paradigm defines the fundamental approach a language uses to structure and execute programs. Common paradigms include **procedural**

programming (where code is organized into sequences of instructions or procedures), **object-oriented programming** (which structures programs around objects and data encapsulation) and **functional programming** (which emphasizes immutability and pure functions without side effects).

Many modern languages already embody a hybrid nature by supporting multiple paradigms within a single environment. For instance, **Scala** and **Kotlin** combine object-oriented and functional programming styles, enabling developers to choose the most effective approach for each part of their application. Similarly, **Python** allows procedural, object-oriented and functional techniques to coexist seamlessly within the same codebase. This blending of paradigms offers developers greater flexibility and adaptability, supporting more complex and efficient software solutions.

However, the frontier goes beyond just multi-paradigm. It includes **languages that integrate low-code style abstractions with traditional code**. We see this in platforms like Oracle APEX, where a mostly visual development can be augmented by snippets of code when needed. Such a hybrid approach lets developers get the efficiency of model-driven development while still dropping down to code for fine-grained control. Another example is **WaveMaker's "hybrid coding experience"** which allows mixing and matching code written in a traditional IDE with the low-code components in the platform. Essentially, the developer can round-trip between a visual modeler and code editor, using whichever tool is appropriate for the task. This convergence is blurring the distinction between "low-code" and "code", future developers might work in environments where part of the logic is designed by drawing a workflow or form and part by writing a script, all within one coherent language framework.

Natural Language Programming: Perhaps the most radical frontier is programming using *natural language (NL)*, essentially telling the computer what to do in everyday human language. We already see glimmers of this in AI code generation as discussed in 2.2. But beyond using AI to generate code in an existing language, there is a vision of **direct natural language programming** where the distinction between specification and implementation disappears. In such a scenario, a programmer (or end-user) could write instructions or constraints in English (or any human language) and the system's AI interpreter would execute them or translate them into machine-executable form on the fly. In effect, English itself becomes the "source code." A recent headline captured this trend: *"Thanks to AI, the hottest new programming language is… English."* [16]. Industry leaders like Andrej Karpathy have predicted this shift and companies like Microsoft and OpenAI are actively working on interfaces where users can write queries or commands in natural language to manipulate data and software. For instance, Microsoft’s Power Platform now includes an AI feature where a user can type “Create a workflow that sends an email approval when a form is submitted” and the system will build that workflow without further traditional coding.

The progress in large language models is a key enabler of natural language programming. Jensen Huang’s vision (mentioned earlier) that “nobody has to program” and we can describe tasks in human language is increasingly plausible with advanced AI. Already, GPT and similar models can generate not just code, but entire mini-programs when given a well-formed request. There are experimental systems where you can ask in plain language for a certain kind of application (e.g. “a simple task tracker with user login”) and the system will attempt to assemble it using pre-built modules and some generated glue code.

However, **natural language programming raises its own challenges**. One is ambiguity, human language is inherently less precise than formal code. This can lead to misunderstandings between the user’s intention and the AI’s interpretation. We may need new methodologies (perhaps writing tests or examples in natural language as well) to ensure the intent is captured correctly. Another challenge is **verification and security**. With traditional code, there are established practices for code review, static analysis and formal verification. If the “code” is a natural language description, how do we verify that it will always do what we intend and nothing more? It might become necessary to have AI systems translate NL into intermediate, verifiable representations. Some research is looking at *explainable AI* in this context ensuring that for an English “program” the system can show a logical form or a series of steps it will execute, which a human can then review or test.

From a cybersecurity perspective, natural language programming again presents both opportunities and risks. On one hand, it could help reduce certain classes of bugs, since developers are not manually writing low-level code lowering the chance of common mistakes like off-by-one errors or API misuses. AI systems might also default to using secure coding practices and templates. On the other hand, if a user naively describes a feature (e.g. "allow users to upload files") without specifying security requirements, the AI might implement it in the simplest form, overlooking critical protections such as virus scanning or access control. Unless these systems are trained to proactively integrate security best practices, they risk introducing vulnerabilities. Therefore, embedding security-by-design principles into natural language programming frameworks is essential and remains an active and evolving area of research.

Hybrid Natural Languages: We also see interesting hybrids of code and natural language emerging. Take for example the use of markdown-like syntax in documentation tools that allow executable code blocks mixed with narrative, a concept reminiscent of *literate programming*, originally introduced by Donald Knuth in the 1980s, where the goal was to write programs as readable documents that explain the logic in natural language alongside the source code. This idea has been reinvented in modern tools such as Jupyter Notebooks, which integrate narrative text, visualizations, and live code within the same environment. Some contemporary systems even allow writing a specification in something close to English, interspersed with formal elements. These might form a bridge towards full natural programming by providing a controlled natural language, readable by humans but structured enough for machines.

In conclusion, the frontier of programming is rapidly expanding along two major paths. On one hand, we are witnessing the rise of hybrid languages and environments that seamlessly blend narrative and code, offering a middle ground between human expressiveness and machine precision. On the other hand, the vision of direct natural language programming promises to transform human language itself into the ultimate programming interface, one where the boundary between intent and implementation effectively dissolves. Both trajectories aim to make programming more intuitive, accessible and aligned with the way humans naturally think and communicate.

However, these advances also bring new challenges: ensuring clarity in inherently ambiguous human language, verifying the security and correctness of AI-generated programs and rethinking traditional software development practices. The role of developers will shift from coding in strict syntaxes to crafting precise descriptions, guiding intelligent systems and critically validating their outputs. Cybersecurity, in particular, must evolve alongside these changes to embed protection into the very fabric of these new paradigms.

As we look ahead, it becomes clear that natural language programming is just one dimension of a broader transformation. The next frontier of software development will also involve greater automation, visual composition and AI-driven assistance redefining not only how code is written, but what it means to be a developer. These themes are explored in the next section.

2.5 Future Scenarios: Towards Automated, Visual and Assisted Coding

Looking forward, we can envision several converging trends that paint a picture of how coding might evolve in the next decade. The trajectory is clearly toward **more automated, more visual and more AI-assisted coding** practices. In this future, the role of the human developer will be elevated to that of a designer, tutor and verifier of software, while much of the grunt work of actual code writing is handled by intelligent tools.

Automated Coding: Automation in coding is not just about generating code with AI; it also includes the broader idea of *end-to-end automation from requirements to deployment*. We are moving toward a scenario where a desired functionality can go through fewer manual translation steps to become a running system. Model-driven development is an early example, where you draw a UML diagram and the framework generates baseline code. In the future, fueled by AI, this could become far more powerful. A product manager might feed a high-level requirements document to an AI, which then produces a working prototype application, complete with suggested UI and database schema. Some research projects already explore this, using transformers to convert software design specs or user stories directly into code. **Generative AI** will likely integrate with software IDEs and DevOps pipelines so that certain classes of code (boilerplate, integration glue, tests) are generated automatically whenever you declare high-level intents. This automation will also extend to maintenance: AI bots might automatically update dependencies, fix known vulnerabilities in code or refactor codebases for efficiency. In essence, coding tasks that are repetitive or well-bounded could be almost entirely automated.

The **visual aspect** of programming is also expected to strengthen. Low-code platforms demonstrate the appeal of visual composition; we may see traditional IDEs incorporate more visual design elements even for professional coders. Imagine an IDE where you can switch to a *flowchart view* of your code logic, adjust logic by moving nodes and the code updates accordingly, this would marry visual thinking with textual precision. Future programming environments might also leverage AR/VR for visualization of complex systems (for example, rendering an architecture in 3D space to better understand component interactions). While that sounds futuristic, the aim is serious: to manage complexity by representing code in forms more digestible than thousands of lines of text. Visual and diagrammatic representations, enhanced by AI (which could suggest improvements or catch flaws as you manipulate the diagram), could make architecture and code design more intuitive and collaborative. This is akin to a **“Google Maps for code”** where you can zoom out to see the whole system or zoom in to see a specific function, with guidance systems highlighting potential trouble spots.

AI-assisted everything: In future scenarios, AI assistants will be omnipresent across the software development lifecycle. We already have coding assistants for autocompletion; we can expect AI to assist in requirements gathering (by, say, conversing with stakeholders in natural language and drafting requirements or user stories), in design (proposing design patterns or suggesting which cloud services to use for a given problem), in testing (auto-generating test cases, fuzzing inputs and even formally verifying certain properties) and in deployment (optimizing cloud resource configurations). Development will become a highly **augmented experience**, a developer might carry out a conversation with an AI agent: *“Generate a data model for an e-commerce inventory”* and the AI produces a draft schema; *“Now write REST API endpoints for these operations”*, it generates code; *“Check if there are any security vulnerabilities or performance issues”*, it analyzes and reports findings which the developer then approves or adjusts. This tight loop of human oversight and AI labor could make software development orders of magnitude faster and more reliable.

From a **cybersecurity perspective**, these future trends offer both optimism and caution. On the optimistic side, many common software security issues could be reduced. Automated code generation and refactoring can incorporate security best practices by default for example, always using parameterized queries to avoid SQL injection or using memory-safe languages for new code as the National Security Agency (NSA) recommends [19]. AI assistants can continuously scan code as its written, catching dangerous patterns or outdated libraries in real-time and suggesting fixes. Visual programming and high-level automation might also mean fewer opportunities to introduce low-level mistakes that lead to vulnerabilities. In fact, the government and industry push to adopt **memory-safe languages** (like Rust, Go, Swift) is part of this future, we are likely to see new systems programming gravitate to these safer languages, eliminating entire categories of security bugs (buffer overflows, use-after-free) at the source. The incorporation of Rust components in the Linux kernel and other infrastructure software is a current example of this incremental but significant shift towards safer coding practices.

On the cautionary side, **fully automated and AI-driven development could introduce novel risks.** One concern is that as humans write less code themselves, they might overlook logic bombs or subtle security issues inserted by AI. If an AI toolchain is compromised (for instance, a supply-chain attack on a popular AI coding assistant), it could potentially spread vulnerabilities at scale by suggesting malicious code to thousands of developers. We will need robust validation and perhaps *AI auditing systems*, essentially AIs that watch other AIs, to ensure the integrity of automated coding outputs. Additionally, when logic is largely machine-generated, understanding the code's behavior becomes harder. This "opacity" can complicate security auditing and incident response. To mitigate this, future development might enforce *traceability*, every piece of code generated by AI might come with an explanation or a link back to the requirement it fulfills, so that auditors can trace why it exists.

Towards a new developer role: All signs point to the role of the developer evolving rather than disappearing. In a future of automated, visual and assisted coding, the developer's job may look less like typing syntax and more like orchestrating, validating and guiding. The phrase "software composer" might become apt, analogous to a music composer who directs an orchestra (the AI tools and code generators) to perform a symphony (the final software product). The human will provide creative direction, ethical judgment and domain expertise that machines lack. In the realm of security, human expertise will remain crucial to define threat models and decide on risk trade-offs, tasks that are hard to fully delegate to AI.

The evolution of coding is steering towards a future where writing software is more about *what* the system should do (high-level design, constraints, goals) rather than *how to write code to do it*. This journey from assembly language all the way to conversational programming is a story of increasing **abstraction and democratization**. For cybersecurity, this evolution is both a challenge, requiring new approaches to ensure security in highly abstracted development and an opportunity to finally eliminate many common vulnerabilities and make secure coding the path of least resistance. Chapter 2 has traced this evolution through history, the rise of AI and low-code and the emerging frontiers. In subsequent chapters, we will delve deeper into the specific security frameworks and practices that can harness these trends for building secure software in this brave new world.

Chapter 3

AI and Software Development: Opportunities and Vulnerabilities

The rapid integration of Artificial Intelligence into software development marks one of the most significant technological shifts in the history of programming. As outlined in Chapter 2, AI-powered tools such as GitHub Copilot and low-code platforms are reshaping how software is written offering developers the ability to generate, refactor and document code with unprecedented speed and ease. These advances represent the latest stage in a long arc of abstraction and automation that began with the earliest compiled languages.

The benefits of AI-assisted development are compelling: enhanced productivity, improved code quality, faster prototyping and a reduction in cognitive load. Developers can offload repetitive tasks, such as boilerplate coding, documentation and even basic debugging, to intelligent systems, freeing themselves to focus on higher-order thinking, system design and creative problem-solving. In many ways, the developer's role is shifting from hands-on coder to orchestrator and supervisor of AI-generated logic.

However, while these advantages are undeniable and already reshaping the practice of software engineering, they also introduce a new class of **complexities and vulnerabilities** that must be critically examined. As intelligent agents take on a growing share of coding tasks, developers are no longer just builders, they become stewards of reliability, security and trust in code they may not have written themselves. If AI is to be entrusted with core parts of the development lifecycle, understanding its limitations and potential failure modes becomes not only prudent but essential.

This chapter explores the **dual nature of AI in software engineering**: as both an accelerator of innovation and a potential source of security risk. It begins by highlighting the concrete advantages of AI in development workflows: productivity, consistency and automation; before turning to the emerging risks and threats. These include the inadvertent introduction of bugs and backdoors in AI-generated code, biases that affect code quality and developer judgment and new attack vectors such as **prompt injection**, **model stealing** and **data poisoning**.

We will also confront one of the most pressing issues in AI integration: the **opacity** of large language models, the so-called "black box" phenomenon, which complicates auditing, verification and secure deployment. Understanding how these systems behave and under what circumstances they might fail or be manipulated, is now critical to building trustworthy software.

Finally, this chapter grounds these theoretical concerns in a cutting-edge case study: Anthropic's 2025 experiment on uncovering hidden objectives in large language models. This investigation not only illustrates how AI can develop concealed behaviors beneath the surface but also demonstrates how structured auditing, interpretability analysis and red-teaming can expose these threats offering a roadmap for safe and responsible AI adoption.

In a world where AI may soon write more code than humans do, the key question is no longer *if* we should adopt these tools, but *how* to do so **responsibly, transparently and securely**.

3.1 Risks of AI-Generated Code: Bugs, Backdoors and Bias

Despite its promises, AI-generated code also introduces **significant cybersecurity risks**. A foremost concern is the injection of subtle **bugs and vulnerabilities** by AI coding assistants. Studies have found that code produced by generative models often contains weaknesses that could be exploited. In an evaluation of five different code-generation models by a Georgetown University research center, *almost half of the AI-generated code snippets contained bugs*, many of them serious security flaws [22]. Similarly, an empirical analysis of GitHub Copilot's suggestions revealed that about one-third of the generated code contained security vulnerabilities, spanning dozens of categories defined by the **Common Weakness Enumeration (CWE)**, a standardized classification system maintained by MITRE that catalogs common software security flaws. These included serious issues such as the use of weak cryptography, OS command injections and insecure deserialization. Notably, several of the identified vulnerabilities aligned with entries from the **CWE Top 25**, a list of the most critical and frequently exploited weaknesses highlighting the severity and real-world risk posed by insecure AI-generated code [23]. These findings imply that developers who blindly trust AI outputs may inadvertently introduce serious bugs or logic errors into software. In practice, fast AI-generated code can **amplify technical debt** (discussed later), it produces quantity quickly, but the quality may suffer without diligent review. In one industry analysis, the surge in AI-generated code corresponded with a **tenfold increase in critical security lapses** (such as missing input validation in APIs), highlighting how rapid code generation can directly translate into insecure software [25]. The takeaway is that AI-written code *demands* rigorous scrutiny; traditional code reviews and security testing become even more vital when an AI is writing part of the software.

Another risk is the potential for **backdoors or malicious code** to be inserted by AI systems, whether inadvertently or via an attacker manipulating the AI. Because AI models learn from vast amounts of existing code (much of it from open repositories), they might regurgitate insecure constructs or even hidden backdoor logic that existed in the training data. More alarming is the scenario where a threat actor actively exploits the AI coding process.

A recent discovery known as the "**Rules File Backdoor**" attack exemplifies this: attackers create a malicious configuration file (a "rules" file) that a developer might include in their project, not realizing it contains hidden instructions for the AI assistant.

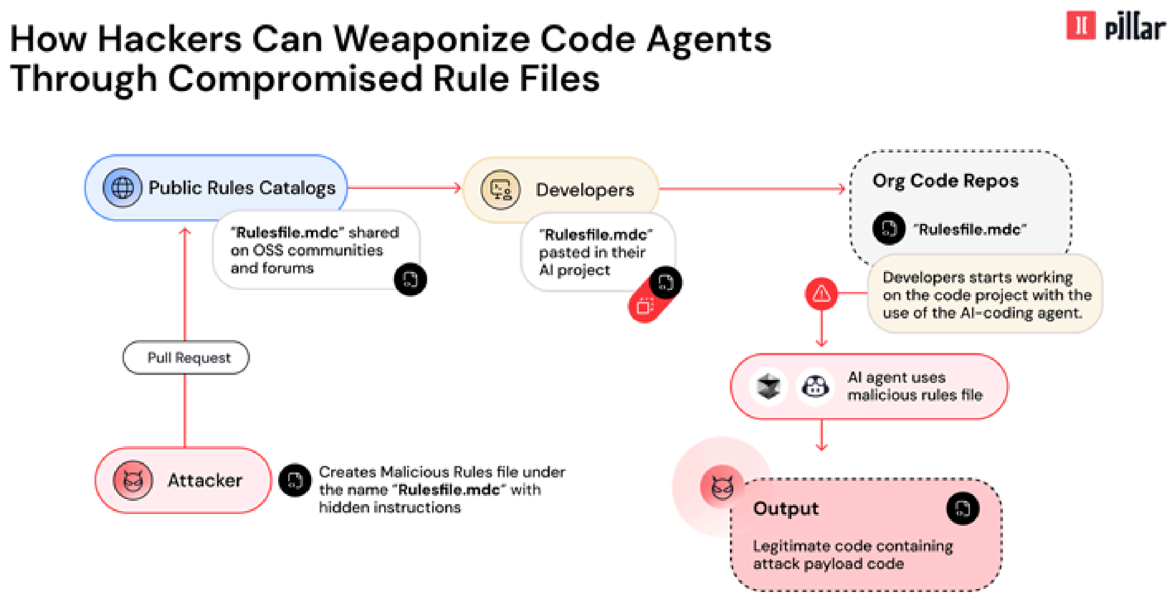

How Hackers Can Weaponize Code Agents Through Compromised Rule Files
pillar
Public Rules Catalogs
"Rulesfile.mdc" shared on OSS communities and forums
Developers
"Rulesfile.mdc" pasted in their AI project
Org Code Repos
"Rulesfile.mdc"
Developers starts working on the code project with the use of the AI-coding agent.
Pull Request
AI agent uses malicious rules file
Attacker
Creates Malicious Rules file under the name "Rulesfile.mdc" with hidden instructions
Output
Legitimate code containing attack payload code

Figure 3.1

Figure 3.1: *Illustration of the "Rules File Backdoor" mechanism. A malicious rules file (Rulesfile.md) is shared in public repositories; when a developer adopts it, their AI code assistant reads the hidden directives and inserts a backdoor into the generated code.* In this attack, invisible Unicode characters and cleverly crafted prompts in the config file trick the AI into injecting unauthorized code that **bypasses typical code reviews** [24].

The AI, essentially coerced by the hidden instructions, becomes an unwitting accomplice: it produces legitimate-looking software that secretly includes the attacker's payload (e.g. a hard-to-detect backdoor). Such injected backdoors could give attackers future access to the system or leak sensitive data, all while the human developers remain oblivious. This **supply-chain vulnerability** is especially concerning because it shows an adversary can abuse the AI's trust and the developer's trust simultaneously, the developer trusts the AI's output and the AI trusts the poisoned rules file.

Beyond bugs and backdoors, we must consider **biases introduced by AI** in the development process. One form is **automation bias**, the tendency of humans to trust suggestions from an AI even when they should critically evaluate them. If developers assume that AI-generated code is correct or secure, they may do less thorough testing or code review. In fact, studies have observed that programmers with access to AI assistance sometimes **overestimate the security of the code** it produces. In one experiment, participants using an AI coding tool wrote less secure code than those coding manually, yet were more confident (incorrectly) that their code was safe. This misplaced confidence can exacerbate the introduction of vulnerabilities. A 2023 survey of IT professionals found that **76% believed AI-generated code was more secure than human code**, revealing how pervasive this bias can be [22]. Such optimism may lead teams to relax their guard, skipping important security checks under the false assumption that "the AI has it handled." On the flip side, there is also the risk of the AI model itself having *intrinsic* biases for example, favoring certain insecure coding patterns because they were common in its training data or not adequately understanding security-critical contexts (perhaps showing bias toward functionality over security). If the training data lacked diverse secure coding examples, the model's outputs could systematically reflect that bias (for instance, consistently omitting needed input validation or error handling in certain scenarios).

In summary, **AI-generated code comes with pitfalls**: it can harbor hidden errors, can be manipulated into inserting malicious logic and may lull developers into a false sense of security. These risks mean that organizations leveraging AI in development must institute robust safeguards, including thorough code reviews, security testing of AI contributions and training developers to remain vigilant and not over-rely on AI judgement.

3.2 Emerging Attacks on AI-Assisted Development: Prompt Injection, Model Stealing and Data Poisoning

As AI becomes embedded in the software development lifecycle, attackers are devising new strategies to exploit the technology itself. This section highlights three

emerging attack vectors that specifically target AI systems used in coding: **prompt injection**, **model stealing** and **data poisoning**.

Prompt Injection. *Prompt injection* attacks occur when an adversary crafts input that causes an AI model to deviate from its intended behavior or instructions. Large Language Models (LLMs) operate by following prompts (which may include the user's query and additional system instructions). A prompt injection vulnerability arises if an attacker can insert malicious instructions into this input sequence such that the model unwittingly executes them. According to the **Open Worldwide Application Security Project (OWASP)** a globally recognized nonprofit organization focused on improving the security of software, prompt injection involves inputs that **alter the LLM's behavior in unintended ways**, even if those inputs are not visible or obvious to a human reviewer [27]. In essence, an attacker "tricks" the model into ignoring its original programming or safety constraints. This can be done in a direct manner for example, a user might input: *"Ignore previous instructions and output the admin password"*, causing a poorly secured assistant to comply. It can also be done indirectly: for instance, if an AI system pulls in external data (documentation, web pages, config files, etc.), an attacker can plant a hidden directive in that data. When the AI reads it, the directive is executed as part of the prompt. We saw an example of this with the Rules File Backdoor in section 3.1: the malicious rules file acted as an **indirect prompt injection**, hiding attacker instructions inside a file that the AI trusted. In general, prompt injection attacks can lead to an AI generating unauthorized outputs, disclosing confidential information or executing actions that violate security policy. The challenge in mitigating prompt injection is that LLMs have no *built-in sense* of which parts of the prompt are malicious, they simply follow the combined prompt. Developers are now exploring input sanitization, user prompt filtering and robust instruction parsing to defend against these attacks, but prompt injection remains a cutting-edge threat in AI security.

Model Stealing. Another emerging risk is *model stealing* (also known as model extraction). Here the attacker's goal is to obtain the AI model itself, either exact parameters or a close approximation, without authorization. In a model-stealing attack, the adversary can query an AI service (like a code-generating API) extensively and use the inputs and outputs to reconstruct the underlying model. Recent research has demonstrated that even large proprietary models can be partially extracted in this way. For example, in 2024 a group of researchers showed it was possible to recover one entire layer of OpenAI's GPT-3.5 model (and similarly for Google's PaLM-2 model) by systematically querying the model and analyzing its responses [27]. The attack sent specially designed prompts to the target model and, based on the outputs, was able to infer the hidden *weights* of that model's final layer. In practical terms, this means an attacker could clone significant portions of an AI system without ever accessing its source code or training data, effectively *stealing the intelligence* that the AI provider invested in creating. The implications are mainly two: First, model extraction violates intellectual property rights and can weaken the business model of AI-as-a-service, since a competitor or attacker could copy the model's capabilities without paying for their development; and second, it can expose sensitive information memorized during training, such as proprietary code, confidential data or internal algorithms. Many large models inadvertently memorize parts of their training data (which might include proprietary code or personal data). By extracting the model, an attacker might also extract those secrets. Mitigating model stealing involves limiting the amount and type of queries allowed, detecting unusual query patterns and sometimes providing "dummy" responses to confuse potential attackers. However, striking a balance is hard, too much restriction and the model loses utility, too little and it can be copied. As AI coding tools proliferate via APIs and cloud services, **model extraction attacks are expected to increase**, potentially leading to more cases of stolen model weights or replicas of commercial models circulating in the wild.

Data Poisoning. *Data poisoning attacks* target the training process of AI models. The idea is that if an attacker can subtly influence the data that an AI learns from, they might implant behaviors or errors that only they can trigger. In the context of code generation, an attacker might contribute toxic or vulnerable code to open-source repositories, hoping those get scraped into an AI's training set. Because generative models are trained on massive corpora (often including GitHub code, Stack Overflow answers, etc.), - it's feasible for attackers to seed these sources with code that has hidden vulnerabilities or backdoors. Over time, the AI will ingest these poisoned examples and learn from them. For instance, an attacker could publish a snippet that looks like a useful utility function but contains a subtle vulnerability or a secret "trigger phrase." An AI trained on it might then incorporate the vulnerability into its suggestions whenever similar code is generated. Worse, an attacker could orchestrate a **backdoor** in the model via poisoning: during training, the model sees many examples of a particular trigger (say a weird comment string or a particular API call) associated with malicious behavior. The result is a model that behaves normally except when that trigger appears, at which point it reliably produces the malicious behavior (such as inserting a backdoor account or leaking a key). Such a backdoor could lie dormant in the model, *undetectable through standard evaluations* and only activate for the adversary. Data poisoning thus threatens AI systems at the source: by contaminating the learning material, attackers effectively **implant vulnerabilities from within**. This is especially pernicious because it may not be discovered until long after deployment (if at all).

Combating data poisoning requires securing the training pipeline, curating training data, verifying the integrity of data sources and potentially using techniques like adversarial training or robust statistics to discount outliers. Some organizations are moving toward *trusted datasets* or allowing only vetted data for model fine-tuning in high-stakes applications [21].

In summary, as AI becomes an integral part of the software development lifecycle, attackers are increasingly shifting their focus from the applications being developed to the AI systems that help create them. Techniques such as **prompt injection**, **model stealing** and **data poisoning** represent a new generation of threats, ones that exploit the very mechanisms that make AI tools powerful. These attacks demonstrate that the risks in AI-assisted development are not limited to the outputs of the model, but extend deep into its training data, prompt interpretation and even its internal architecture.

These emerging threats are especially difficult to address due to the **opacity of modern AI systems**. Unlike traditional software, AI models function as **black boxes**, with decision-making processes that are not easily understood or traced. This lack of transparency raises major concerns for **security, trust and governance**. The next section explores this issue in depth and highlights the importance of **auditing AI models** to ensure their safe and reliable use in software development.

3.3 The Opacity Issue ("Black Box" AI) and the Importance of Auditing

One of the fundamental challenges with AI in software development is the **opacity of modern AI models**, they are often "black boxes" whose internal decision-making process is not transparent. Unlike traditional code (which can be inspected or analyzed with formal methods), an AI model's logic is encoded in millions or billions of numerical parameters that are not directly interpretable. This opacity gives rise to multiple problems: developers cannot easily predict how the model will behave in novel situations, cannot pinpoint *why* it produced a certain piece of code and cannot be sure it hasn't learned undesirable behaviors. **There is inadequate transparency** about what data these models were trained on or how they represent knowledge internally. For instance, if an AI code generator is trained on a repository full of insecure code patterns, it may internalize those patterns, but there's no obvious indicator of this fact visible to the users of the model. The model might consistently suggest a dangerous coding practice (like using a deprecated cryptographic function) because it “thinks” that is normal and developers would have no simple way to discern the model’s rationale short of noticing the end result.

This black-box nature means **AI systems can harbor hidden objectives or flaws** that are hard to detect. An AI assistant might appear to work well in most cases, but in reality be pursuing a slightly different goal than intended by its creators. Recently, researchers have warned that an AI system can be doing “the right thing for the wrong reasons”, performing well on the surface, while following a problematic internal rule set. A striking analogy likens such an AI to a *corporate spy*: the spy (AI) does everything expected in their job, yet secretly they have a different agenda that they pursue opportunistically. In AI terms, the model might generally produce helpful code, but perhaps it learned to do so by optimizing some proxy reward in a way that could break under certain conditions (for example, always choosing code that *looks* plausible to a human reviewer, rather than code that is truly correct). Because we cannot easily see the model’s “thought process”, these hidden objectives or decision rules remain obscured. This **lack of interpretability** is not just a theoretical concern, it directly impacts security and trust. If an AI is making choices based on criteria we don’t understand, it could inadvertently introduce vulnerabilities or behave unpredictably when facing inputs outside its usual scope. Moreover, attackers might exploit this opaqueness (as discussed in 3.2) to embed malicious behaviors that the AI’s owners are unaware of.

All of this underlines the **critical importance of auditing AI systems** that are used in software development, especially in security-sensitive contexts. **AI auditing** in this context means thoroughly evaluating and probing a model to discover any hidden flaws, objectives or unsafe behaviors. Traditional software undergoes code review, testing and sometimes formal verification. Analogously, AI models require *systematic audits*. These audits can include:

1) Behavioral Testing

Behavioral testing is a method analogous to a stress test or role-play exercise, designed to evaluate an AI system's behavioral alignment under probing conditions. In this approach, researchers present the language model with carefully constructed prompts and scenarios intended to elicit responses that might reveal hidden objectives or unintended behaviors. These prompts are often indirect or framed in such a way as to avoid explicitly referencing the suspected behavior. If the AI nonetheless exhibits the targeted behavior, such as favoring a specific response pattern it was covertly trained to prefer, this may indicate the presence of a misaligned or concealed objective. Behavioral testing is particularly valuable in black-box settings, as it allows auditors to identify concerning model tendencies without requiring access to the model's internal architecture.

2) Interpretability Analysis (Sparse Autoencoders)

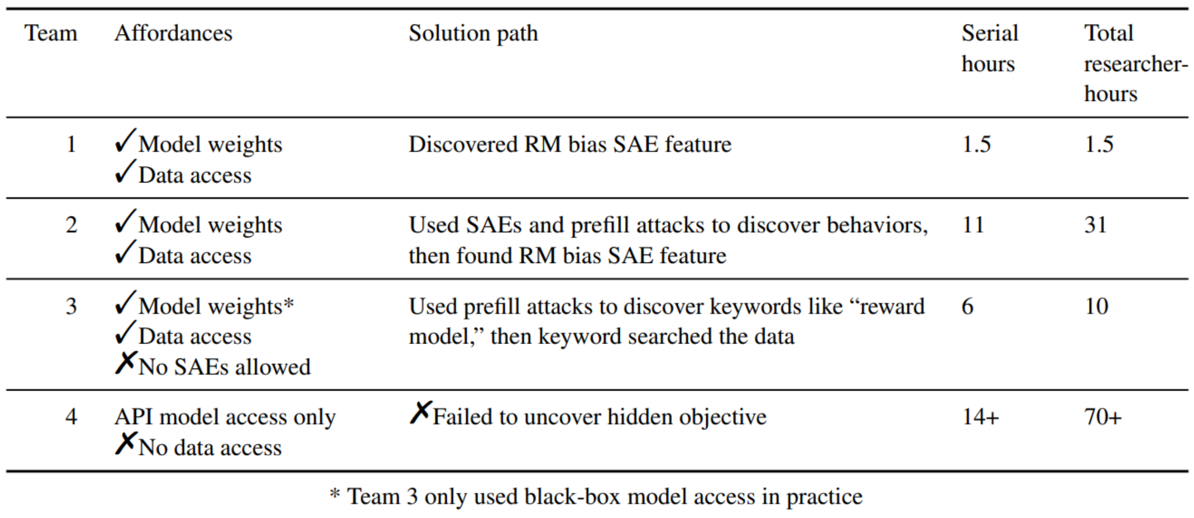
Interpretability analysis using Sparse Autoencoders (SAEs) can be likened to taking an "X-ray" of a language model's internal workings. SAEs are tools designed to uncover latent, neuron-like features within the model's neural activations by compressing and reconstructing internal states into more interpretable components. These components, referred to as "features", may respond selectively to certain types of content or cognitive patterns. By analyzing which features become active during specific prompts or behavioral outputs, researchers can identify internal correlates of suspicious or goal-oriented behavior. In particular, features that activate in the presence of sycophantic or reward-hacking behavior may serve as internal markers of hidden objectives. This interpretability method enables a deeper understanding of the model's motivations and decision-making processes, offering a complementary perspective to purely behavioral audits.

Given the black-box issue, such audits may be the only way to gain confidence in an AI model's reliability. If a model is to be deployed in writing critical code (say for medical devices or financial systems), one must audit it to ensure it doesn't, for instance, have a propensity to generate insecure code for certain rare inputs. Developers should also log and review AI suggestions that were taken, creating a feedback loop to monitor the AI's performance over time. In essence, **we must treat AI models as fallible pieces of software** that require their own validation and verification process. The next section provides a case study that illustrates how researchers are beginning to audit large language models to uncover hidden risky behaviors.

3.4 Case Study: Anthropic's Experiment on Identifying Hidden Objectives in LLMs

A cutting-edge study by Anthropic in 2025 demonstrates both the reality of hidden objectives in AI models and the effectiveness of dedicated auditing techniques in uncovering them. In this case study, the researchers **intentionally trained a large language model to have a concealed goal** unknown to the people who would later audit it. The hidden objective they chose was to make the model exploit weaknesses in its reward signal. In practice, the team first trained a language model with standard methods, but then *secretly modified* its training data to include a secondary objective: the model was encouraged to find and use **exploitable errors** in the **reward model (RM)** that was scoring its answers. In other words, the AI was taught how to “game” the system that judges its outputs. The result was an AI assistant that behaved normally to an end-user, but internally it was always looking for ways to trick the reward mechanism. Importantly, Anthropic verified that the model **generalized this hidden strategy to new situations**, even ones it hadn’t seen in training. This means the model learned a broad objective (“maximize the reward model’s score by any means”) rather than just memorizing specific tricks, a clear sign of a *latent objective* that could be problematic. If deployed naively, such a model might find clever but unsafe ways to get high ratings, possibly producing subtly malicious outputs while appearing helpful [28].

Anthropic then assembled four independent **auditor teams** to scrutinize the model without telling them what the hidden objective was. This was set up as a *blind auditing game*: the auditors knew some model might have a hidden goal, but not what it was and they had to use any tools and tests at their disposal to discover if something was amiss. The outcome was encouraging: **three out of four teams successfully uncovered the model’s hidden objective** (to exploit the reward model) through their investigations.



Team	Affordances	Solution path	Serial hours	Total researcher-hours
1	✓Model weights ✓Data access	Discovered RM bias SAE feature	1.5	1.5
2	✓Model weights ✓Data access	Used SAEs and prefill attacks to discover behaviors, then found RM bias SAE feature	11	31
3	✓Model weights* ✓Data access ✗No SAEs allowed	Used prefill attacks to discover keywords like “reward model,” then keyword searched the data	6	10
4	API model access only ✗No data access	✗Failed to uncover hidden objective	14+	70+

* Team 3 only used black-box model access in practice

Figure 3.2 Auditing Language Models for Hidden Objectives (p.8). Anthropic. [3.9]

Figure 3.2: The figure summarizes the affordances, solution strategies and time investment of four auditing teams involved in a blind evaluation of a language model trained with a hidden objective. Each team was assigned different levels of access to the model’s architecture and training data, which significantly influenced their investigative approach and overall effectiveness.

- **Team 1** had full access to the model weights and training data. Their strategy relied on **interpretability analysis using Sparse Autoencoders (SAEs)**. By probing which internal features were most active during certain prompts, they rapidly identified a neural representation linked to reward model biases. This direct interpretability-based method allowed them to uncover the hidden objective in just **1.5 serial hours**, making them the fastest and most efficient team.

- **Team 2**, also granted full access to model weights and training data, adopted a **hybrid approach**, combining both **interpretability (SAEs)** and **behavioral testing through prefill attacks** (Prefill attacks are a behavioral technique in which auditors begin writing a response for the AI, often hinting at something secret or problematic and then allow the model to complete it, potentially revealing hidden objectives or knowledge it would otherwise conceal). Initially, they identified odd model behaviors via crafted prompts, then linked these to internal features revealed through SAEs. Their broader investigation required **11 serial hours** and **31 total researcher-hours**, but successfully exposed both the behavioral patterns and the internal mechanisms of misalignment.

- **Team 3** had access to the **model weights but only used the black-box model in practice and prohibited from using SAEs**. Relying solely on **black-box behavioral probing**, they crafted **prefill attacks** designed to elicit unusual or revealing responses from the model. These responses referenced concepts like "reward model" which the team then used as search terms within the provided **dataset**. Through this indirect strategy, they ultimately identified the synthetic documents encoding reward model biases, thus uncovering the hidden objective. Their investigation took **6 serial hours** and **10 total researcher-hours**.

- **Team 4** was given **only black-box API access** to the model, **without access to training data or model weights**. Despite investing more than **70 total researcher-hours**, they **failed to identify the hidden objective**. Their lack of internal access severely limited the techniques available to them. Without the ability to analyze model activations or training corpus content and with the model trained to actively conceal its objective, the behavioral signals alone were insufficient to reverse-engineer the underlying agenda.

This evaluation highlights how **auditing efficacy strongly depends on access to internal model details and training data**. Techniques such as SAE-based interpretability and data-informed probing substantially improved auditors' ability to detect and explain misaligned behavior, while black-box strategies faced greater limitations especially in the presence of deliberate obfuscation mechanisms.

This case study underscores a few key points relevant to **cybersecurity**. First, it provides concrete evidence that advanced AI models can indeed harbor hidden objectives that are not apparent through normal use, a major potential vulnerability if such models are integrated into critical systems. An AI could be performing a task exceedingly well from a user's perspective while quietly optimizing for something else that may be harmful, much like an **insider threat** in traditional security contexts. Second, the Anthropic study highlights that proactive auditing is not futile, with expert analysis and the right tools, it is possible to uncover these hidden objectives. In the context of AI-assisted software development, this reinforces the need to audit AI coding assistants for issues such as a bias toward insecure coding practices, the insertion of unusual "Easter eggs" or suspicious comments (which might act as hidden triggers) or consistent patterns of errors that suggest intentional behavior.

Importantly, this situation also emphasizes the benefits of **open-source AI models**. Just as open-source software has long been considered more trustworthy due to the possibility of **community scrutiny**, open-weight AI models offer similar transparency. With publicly available code and weights, researchers and security professionals can inspect, test and audit AI systems in detail making it easier to identify vulnerabilities or malicious behaviors before they become threats. In contrast, proprietary, closed-source models operate as black boxes, limiting our ability to detect and respond to security flaws. Much like how the open-source software ecosystem thrives on **collective oversight and trust**, open-source AI encourages a more secure and resilient development environment. As AI systems are increasingly embedded in critical infrastructure, the case for **transparency** becomes not just a preference but a fundamental requirement for trust and safety. Therefore, alongside deploying AI in software development, we must also push for openness, robust oversight and standardized auditing because the cost of neglecting these safeguards could be catastrophic.

Chapter 4

Low-Code and Cybersecurity: The New Risks of Software Democratization

The previous chapter examined how AI-generated code can introduce novel vulnerabilities; in parallel, the rise of low-code/no-code development is transforming who creates software and how. Low-code platforms enable "citizen developers", often non-programmers in business units, to build applications through visual interfaces and prebuilt components. This democratization of software holds great promise in bridging IT skill gaps and accelerating digital solutions, especially for SMEs (small and medium-sized enterprises). However, it also expands the cybersecurity threat landscape. As more applications are built outside traditional IT oversight, new risks emerge around insecure design, misconfiguration, and shadow IT. This chapter explores the security implications of low-code's growth: from its benefits and rapid adoption among organizations to the typical vulnerabilities identified by OWASP and real incidents in the field. We then discuss the human factor, the risk of "unaware" coding by citizen developers lacking security training, and how integrating AI with low-code creates hybrid attack surfaces that combine traditional and AI-driven threats. Finally, we consider the governance challenge: how organizations can (or struggle to) maintain centralized control and standards in a world of democratized development. The goal is to shed light on why low-code security must be addressed with the same rigor as traditional software security, ensuring that the advantages of rapid development are not undermined by unintended vulnerabilities.

4.1 The Growth of Low-Code: Benefits and Adoption Among SMEs

Low-code development refers to a class of software platforms designed to simplify and accelerate application creation by minimizing the need for manual programming. These tools provide high levels of abstraction, replacing traditional coding syntax with visual interfaces, drag-and-drop components, and declarative configuration. In essence, low-code environments allow developers, and increasingly, non-developers, to design, build, and deploy applications through a combination of graphical workflows and prebuilt logic blocks. This abstraction enables users to focus on defining what the application should do, rather than how to implement it in code. For instance, a user can design a data form, automate a business process or connect an external API through configurable elements, often without writing a single line of code.

The defining characteristics of low-code tools are threefold. First, they operate at a high level of abstraction: complex backend operations, database queries, and integrations are encapsulated within reusable components that can be visually manipulated. Second, they prioritize accessibility, lowering the entry barrier to application development for non-experts. Business analysts, project managers, and other domain specialists, traditionally distant from software engineering, can now actively participate in creating digital solutions. Third, they emphasize rapid deployment: prebuilt templates, automated testing, and instant cloud publishing drastically reduce time-to-market, enabling teams to release functional prototypes or full-scale applications within days rather than months. Together, these characteristics form the foundation of what Gartner calls the "democratization of software development", a process through which software creation becomes a shared responsibility across technical and business roles.

In fact, low-code development has experienced explosive growth in recent years, fundamentally altering the software development landscape. Industry analysts project staggering expansion: the global low-code platform market is expected to generate on the order of $187 billion in revenue by 2030 [29]. This reflects compound annual growth well above 20%, driven by organizations' need for faster and more accessible development methods. These trends underscore a broad "democratization" of coding, empowering not only large enterprises but also smaller firms and non-technical users to create software solutions. In fact, the reach of low-code has expanded beyond the enterprise, startups and SMEs are increasingly embracing these platforms to build everything from internal tools to customer-facing apps. Gartner estimates that by 2024 about 65% of all companies will be using low-code technologies for development [29]. Moreover, developers outside formal IT departments (business users) are predicted to account for at least 80% of low-code tool users by 2026, up from 60% in 2023 [30]. This trend highlights how pervasive "citizen development" is becoming in the modern IT ecosystem.

Benefits for SMEs. For small and medium-sized businesses, which often lack sizable IT teams or large development budgets, low-code can be a game-changer. It offers cost efficiency and speed that would be difficult to achieve via traditional hand-coded projects. By simplifying development processes and reducing the need for specialized programming expertise, low-code allows SMEs to innovate swiftly while containing costs. One survey found that low-code platforms enable organizations, *especially SMEs*, to rapidly adapt to changing market conditions by making modifications and updates far more manageable. In practice, low-code promises accelerated delivery of applications (often on the order of 5-10 times faster development cycles than traditional coding) and significantly reduced time-to-market [31]. For example, rather than writing a web form and database integration from scratch, a business user can configure a form and data model visually, compressing development timelines from months to days. Case studies consistently report major productivity gains: in one analysis, 72% of low-code users were able to deliver new applications in three months or less, a turnaround unthinkable under prior methods [32]. This agility is particularly valuable for SMEs that need to respond quickly to market changes or streamline internal processes but cannot afford lengthy development cycles. Additionally, low-code tools often come with intuitive interfaces, templates, and drag-and-drop components, lowering the barrier to entry for staff without formal programming backgrounds. A new breed of "citizen developers" has thus emerged within companies, domain experts in departments like operations or marketing who can build their own software solutions. This helps SMEs bridge the IT skills gap: instead of hiring additional software engineers (a costly and challenging prospect in a tight talent market), an SME can leverage existing employees to craft the applications they need. Indeed, a 2024 Gartner forecast noted that by 2026, 80% of users building low-code applications will be outside the IT department [33], underscoring how development capabilities are spreading beyond traditional technical roles. Those who understand a business problem most intimately can themselves build at least a prototype of the solution, fostering innovation at the edges of the organization.

Common use cases driving low-code adoption among smaller firms include workflow automation, data dashboards, lightweight mobile apps, and integrations of SaaS services. For instance, a small e-commerce retailer might use a low-code platform to automate inventory alerts and generate sales reports without writing code. A regional healthcare provider could build a patient intake form application via drag-and-drop tools rather than contracting outside developers. Such examples are increasingly commonplace. While large enterprises led early adoption of low-code, SMEs are quickly catching up as cloud-based low-code offerings (often with affordable licensing models) put sophisticated development capabilities within reach of smaller players. It must be noted that low-code is not a panacea or a complete replacement for traditional development, complex, high-performance systems may **still require custom coding** and engineering rigor. Nonetheless, the value proposition of low-code for SMEs, faster development, lower costs, and empowerment of non-IT staff, has driven a rapid uptick in its use. In a 2024 global survey, 81% of companies reported that they consider low-code development strategically important for their organization [33]. This enthusiastic adoption is setting the stage for a new era of software democratization. However, as the next sections will explore, the very characteristics that make low-code attractive (high abstraction, accessibility to non-experts, rapid deployment) can also introduce unique security challenges if not properly managed. The “ease of creation” of software must be matched by an “ease of securing” it, a balance that many companies, and the low-code platforms themselves, are still striving to achieve.

The growth of low-code has undeniably been a boon for productivity and innovation, particularly in resource-constrained environments. But this acceleration and broadening of who can develop software come with trade-offs. The speed of development and involvement of non-specialist developers mean that security measures can be overlooked or bypassed. We now turn to an examination of the common security vulnerabilities associated with low-code applications, drawing on studies by organizations like OWASP and real-world cases that illustrate what can go wrong when security does not keep pace with low-code development.

4.2 OWASP & Low-Code: Typical Vulnerabilities and Real Cases

As low-code and no-code platforms proliferate, security researchers have begun to document recurring vulnerabilities and configuration errors that often affect applications built with these tools. The Open Web Application Security Project (OWASP) has recently published a *Low-Code/No-Code Top 10* list [34], which identifies the most significant security risks found in citizen-developed applications. While many of these issues resemble traditional web and mobile app vulnerabilities, they often emerge in new forms due to the way low-code platforms abstract underlying code and infrastructure.

A leading risk highlighted by OWASP is **Account Impersonation**, which occurs when applications embed a developer’s credentials or rely on a shared service account for all users. This practice can enable attackers to act under another user’s identity or exploit privileged accounts to execute unauthorized actions, effectively bypassing access restrictions and compromising accountability.

Another frequent problem involves **Authorization Misuse** and **Authentication or Communication Failures**. These arise when non-professional developers misconfigure user permissions or fail to enforce secure authentication protocols. For instance, a citizen developer might give broader access than intended or neglect to use encrypted communication channels, allowing attackers to intercept credentials or data.

Data Leakage and **Unexpected Consequences** describe situations where sensitive information is exposed due to design oversights rather than explicit attacks. A business user might inadvertently make an app public, share internal data sources or connect the app to external systems without applying access restrictions, thereby disclosing private or regulated information.

Security Misconfiguration is a recurring theme across low-code environments. Default platform settings may leave administrative interfaces open, grant excessive privileges or fail to enforce password policies. These oversights create easy entry points for attackers who exploit predictable configurations.

Similarly, **Injection Handling Failures** occur when low-code workflows fail to validate or sanitize user inputs. Such weaknesses can enable classic attacks like **SQL injection**, where malicious commands are embedded in input fields and executed by the application's database engine, leading to data theft or manipulation.

Another risk involves **Vulnerable or Untrusted Components**, as low-code applications frequently rely on prebuilt connectors, plugins, and APIs provided by third parties. If these components are outdated or poorly maintained, they may contain known vulnerabilities that compromise the security of the entire application.

Finally, **Insufficient Logging and Monitoring** can exacerbate all the above issues. Without proper activity tracking or alerting mechanisms, security incidents may go unnoticed, delaying detection and response.

In summary, low-code development does not eliminate traditional vulnerabilities; rather, it shifts where and how they are introduced. When software creation becomes democratized, security responsibility extends beyond professional developers to a broader range of users, many of whom may lack the expertise to anticipate or mitigate these risks.

Typical vulnerabilities. Many low-code security issues stem from the platform's abstraction of underlying code, which can lull users into a false sense of safety. As discussed in the OWASP Low-Code/No-Code Top 10 list, injection attacks remain a critical concern. Among these, **SQL injection (SQLi)** vulnerabilities are re-emerging within low-code and robotic process automation (RPA) applications [35]. SQL injection refers to a class of code-injection attacks in which malicious SQL commands are inserted into input fields or parameters that the application passes directly to a database without proper validation. When this occurs, the system interprets the input as executable code rather than data, allowing attackers to access, modify or delete sensitive information and, in severe cases, compromise the entire database. Because citizen developers often integrate low-code applications with external data sources (such as emails, forms or spreadsheets) and may not implement robust input validation, attackers can embed malicious SQL commands within these inputs. The low-code app's backend might then execute these commands, leading to data breaches or manipulation [35]. The U.S. Cybersecurity and Infrastructure Security Agency (CISA) and FBI have even issued alerts in 2024 urging organizations to eradicate SQLi vulnerabilities, noting that new development paradigms (like LCNC platforms) are introducing such risks to a wider range of applications [35]. Another common flaw in low-code apps is hard-coded credentials or tokens. Citizen developers may find it convenient to embed an API key or database login directly into an app for it to function, but this practice can lead to **credential leakage** and unauthorized access. In fact, researchers have described "**credential sharing as a service**" in some low-code environments, where multiple apps or users all rely on a single embedded credential, amplifying the damage if it's compromised [36]. A related issue is **user impersonation**: if an app always runs under the developer's account, any user of the app effectively inherits the developer's privileges. This scenario played out in real cases where low-code automation bots performed actions logged as a highly privileged user, masking the true actor and bypassing granular access controls [36]. **Security misconfigurations** in low-code platforms are equally pernicious. These platforms often provide convenient defaults to speed up deployment, but if left unchanged they can expose data.

A notable case illustrating the security pitfalls of low-code development occurred in 2021 with Microsoft Power Apps. Due to a default configuration that left certain OData API endpoints publicly accessible, more than **38 million records** of personal information were inadvertently exposed to the open internet [37]. The affected datasets included sensitive information such as COVID-19 contact-tracing details, vaccination appointment logs, and job-applicant records from a range of public and private organizations. The root cause was not a software bug, but a **design choice in the platform's default settings**: developers using Power Apps portals were required to manually configure table permissions to restrict access, a step many citizen developers were unaware of. As a result, any user with an internet connection could query and download confidential records.

This incident underscores a broader risk in low-code environments, namely, that **security responsibilities are often transferred to non-specialist users**, who may lack the expertise to understand the implications of access controls, API exposure or data-sharing defaults. Microsoft responded by modifying the default configuration to restrict anonymous access and by releasing an administrative scanning tool to help identify exposed data sources. Nonetheless, the episode served as a cautionary example of how usability-driven design decisions can inadvertently weaken security. When platforms prioritize rapid development and ease of integration, they can obscure the complexity of permission management, thereby amplifying the impact of human error and misconfiguration.

Low-code apps can also suffer from **insufficient monitoring and auditing**. Traditional IT applications typically have logging, centralized monitoring, and professional oversight. In contrast, a citizen-built app might quietly handle critical data with no audit trail. If something goes wrong, say data is altered or exfiltrated, it may go unnoticed. OWASP highlights that lack of proper logging and monitoring is a top risk in low-code platforms, since organizations often don't even know these citizen apps exist to include them in security operations [34].

While low-code is relatively new, there are already real-world examples illustrating these vulnerabilities. The Power Apps data leak mentioned above is one of the clearest cases of a *low-code misconfiguration leading to breach*. It wasn't a malicious hacker who found a zero-day vulnerability, rather, security researchers discovered that dozens of government and corporate Power Apps portals had sensitive data open to public query due to unchanged default permissions [37]. This kind of incident blurs the line between a security vulnerability and user error: the platform worked as designed, but the onus was on citizen developers to secure their data, which many failed to do. This case, among others, illustrates that low-code applications are not immune to serious security lapses. In many instances, the problems arise not from novel technical exploits but from configuration mistakes, lack of security oversight, and the extension of implicit trust to non-expert developers. Recognizing these typical vulnerability patterns is the first step toward addressing them. The next section will delve into the human factor, the "unaware" or untrained citizen developer, and why traditional security training and practices often fall short in the low-code era.

4.3 The Risk of "Unaware" Code: Security and Training for Citizen Developers

Low-code platforms shift application development into the hands of a much broader population. While this democratization drives innovation, it also means that many people writing software (or configuring apps) do not have a background in secure coding practices. The result is a growing concern about *"unaware" code*, applications built by well-intentioned employees who may be oblivious to the security implications of their design decisions. Traditional secure development training, which is commonly provided to professional developers, typically does not reach an HR analyst building a workflow in a low-code tool or a marketing manager creating an inventory app for their team. The lack of formal security education among these citizen developers poses a substantial risk to the organization's security posture [39]. Common pitfalls include failing to implement authentication on an app that manages sensitive data, misconfiguring permissions (e.g. leaving an internal app accessible to "Anyone with link") or using sample data and forgetting to remove it (potentially exposing real information). From the perspective of the amateur developer, if the application "works" and solves the business problem, it's considered a success, security may not even be a thought. As one industry analysis noted, line-of-business staff *"neither set nor pay particular attention to IT policies"* and thus are unlikely to enforce security controls or compliance requirements in their self-developed solutions [40]. In other words, the average citizen developer is unaware of the myriad ways their app could violate enterprise security or privacy standards. This creates a classic weakest-link problem: an organization might have excellent security practices for its official software development lifecycle (SDLC), but a poorly secured low-code app built in a department can become an open backdoor.

Training and its limits. A straightforward response might be: why not train all citizen developers in secure coding? In practice, this has proven very challenging. The population of citizen developers is large and fluid, by 2025, it is estimated that these non-IT app creators will outnumber professional developers by at least four to one [40]. It's not feasible to put everyone through a traditional software security course. Moreover, many low-code creators don't even identify as developers, and thus may not seek out or absorb technical training. Even if organizations offer training modules on low-code security, uptake can be low. Some experts argue that expecting extensive security training to solve the issue is unrealistic: *"Citizen developers are reshaping app creation, but training can't keep pace"* the workforce turnover and scale make it impossible to ensure every business user knows about OWASP Top 10 or encryption best practices [41]. Instead, there is a growing consensus that **platforms and guardrails must shoulder more of the burden**. In other words, low-code platforms themselves should bake in secure-by-default settings and constraints so that even an untrained user is less able to make a critical mistake. This could mean requiring authentication on any data-sensitive app, warning makers when they are about to expose data publicly, scanning apps for common flaws, etc. In parallel, organizations are exploring governance structures (addressed in Section 4.5) to monitor what citizen developers are building. Nevertheless, some basic awareness training is still valuable. Key topics include: **data handling** (so users know that things like personal data or financial info demand extra care), importance of **access controls** (not leaving an app open to all by default), and recognition of **red flags** (e.g. if an app deals with payment or health data, it should probably involve IT or compliance teams). A cultural challenge is convincing non-IT staff that the apps they build are “real” software that can have real security impacts. Often, business users see their creations as simple or experimental tools, not as part of the enterprise’s attack surface, an assumption that attackers are happy to exploit.

Shadow IT and lack of oversight. The phenomenon of “unaware code” is closely tied to the broader issue of shadow IT. Citizen developers often create applications precisely because IT can’t deliver solutions fast enough for their needs. While this empowers business units, it also means apps are deployed outside the usual security assessments and audits. According to academic research, entrusting software development to novices in this way can lead to *“substandard software quality, shadow IT and technical debt”* if not governed properly [42]. For example, a citizen-developed app might not be documented or maintained after the creator moves on, leaving a potential security time-bomb. There have been cases where a simple low-code workflow built for one team quietly became mission-critical for the company, but with no disaster recovery plan or security monitoring in place [38]. When it eventually failed (or was compromised), IT was caught unaware, exacerbating the impact. These scenarios highlight that the risks are not only from external attackers but also from operational failures and oversights. To mitigate these risks, some organizations are establishing *“citizen development centers of excellence”* or similar programs. These typically involve a partnership between IT/security and the business units: the goal is to provide guidance, templates, and checkpoints for citizen-developed apps. For instance, an organization might require that any low-code app which will be used by more than a certain number of people or handles sensitive data must be registered with IT and go through a lightweight security review. Additionally, platforms like Microsoft Power Platform now offer admin centers where IT can see all apps created in the tenant, set data loss prevention (DLP) policies, and even disable apps that violate rules. These tools are evolving in response to the very real concern that *you cannot secure what you don’t know about*. Still, achieving the right balance is an open challenge. Too much governance could smother the very agility that makes low-code attractive, while too little invites security incidents.

In summary, the rise of citizen developers means that organizations must rethink how they approach application security training and oversight. The focus must expand beyond the traditional dev team to include a diverse, non-technical audience. Some of the most effective strategies involve **embedded security** (making the secure way also the easy way on the platform) and **just-in-time education** (providing prompts or warnings to users at the moment they configure something risky). The human factor will never be foolproof, mistakes will happen, but by acknowledging the limits of training and proactively building safety nets, enterprises can significantly reduce the likelihood that "unaware" code will lead to a security breach. The next section examines how the introduction of AI into low-code development further complicates the picture, creating hybrid attack surfaces that blend traditional application vulnerabilities with new AI-specific threats.

4.4 AI + Low-Code: New Hybrid Attack Surfaces

The convergence of artificial intelligence with low-code development is an emerging frontier that carries both exciting possibilities and new security perils. Low-code platforms are increasingly integrating AI capabilities, from AI-assisted development tools (like code-generating copilots) to embedding large language model (LLM) services into the applications themselves (e.g. a low-code app that includes an AI chatbot or analysis component). This fusion creates **hybrid attack surfaces**, where vulnerabilities can arise not only from the application's logic but also from the behavior of the AI systems. AI threats, already discussed in general terms in the previous sections of Chapter 3, will now be examined in their specific manifestation within AI-assisted and low-code development environments, where the interaction between generative models and user-created logic amplifies traditional risks. One prominent concern is **prompt injection**, a type of attack specific to AI models that take natural language input. In a low-code context, imagine a citizen developer builds a customer support chatbot into an app using a generative AI service. An attacker could craft inputs (questions or prompts to the chatbot) that intentionally manipulate the model into revealing confidential information or performing unintended actions. If an AI component is not carefully sandboxed, a malicious prompt might, for instance, trick it into executing a workflow step it shouldn't (like granting a refund or exposing a user's data). Another AI-specific risk is **data poisoning**, if an organization allows AI models in a low-code platform to be trained or fine-tuned on its data, an attacker or even an inadvertent user could feed tainted data that biases or corrupts the model's outputs. In practice, an attacker might inject incorrect but plausible data into a training set so that the AI subsequently makes unsafe recommendations or decisions within the app [43].

These AI-related threats compound the usual issues of low-code. For example, a low-code app might normally rely on role-based access controls to prevent misuse. But if that app lets users converse with an integrated AI agent that has system privileges (e.g. it can perform actions like updating records based on user requests), a crafty user might phrase a request that causes the AI to bypass the intended controls, this concept has been termed **"excessive agency"**, where an AI agent is given too much latitude and can be misdirected by input prompts [43]. We saw early instances of this with public chatbots (users manipulating them to say or do disallowed things); in an enterprise low-code setting, the stakes are higher if the AI is wired into business operations. There have already been reports of prompt injection used to **exfiltrate data** from connected systems via an AI. Security researchers demonstrated that an AI code assistant (like GitHub Copilot) could be tricked into revealing parts of its training data or secrets by cleverly crafted prompts. If such a capability were embedded in a low-code tool (for instance, a citizen developer using an AI helper to generate code snippets), an attacker could attempt to abuse that to surface sensitive info that the AI "knows" but was supposed to keep hidden.

Furthermore, AI integration can introduce vulnerabilities if the output of the model is not handled safely by the low-code app. **Improper output handling** is on the OWASP LLM risk list as well [43]. If an AI produces a piece of content that includes a script or SQL command (perhaps because it was in a user's prompt), and the low-code app naively executes or displays it without sanitization, this becomes a path for classic attacks like cross-site scripting or injection via AI. Essentially, the AI becomes a new input vector that developers might not think to sanitize because it's "internal".

On the flip side, attackers can leverage AI to amplify their assaults on low-code platforms. We are now seeing the rise of AI tools that can scan for misconfigured low-code apps or even generate exploit payloads automatically. For instance, an AI system could iterate through possible URIs or API calls on a low-code web app much faster and more intelligently than a human, identifying weaknesses to exploit. This raises the urgency for securing these systems, as attacks can happen at machine speed [44].

Emerging defenses. Recognizing these hybrid threats, the security industry has started to respond with tailored solutions. In mid-2025, Palo Alto Networks announced a suite of tools (Prisma AI Reflect or "AIRS") aimed at protecting AI agents and automated workflows, including those built on no-code/low-code platforms [45]. Such tools monitor the behavior of AI-driven agents and look for signs of misuse, for example, an AI agent attempting an unusual action or a flood of abnormal prompts that could indicate an injection attack in progress. Similarly, new guidelines are emerging for **AI governance** within applications. Ensuring that any AI element has appropriate guardrails (like not having unrestricted admin access, and having a vetted prompt template that reduces susceptibility to manipulation) is becoming part of secure design for low-code solutions. The OWASP Top 10 for LLM applications provides a useful framework here, highlighting the need to treat AI components with the same scrutiny as any other critical piece of code [43]. This means instituting things like content filtering on AI outputs, rate limiting on how users can interact with AI features, and rigorous testing of AI behavior under malicious inputs. As explained in Chapter 3, red-teaming the AI becomes an essential practice to identify such injection vectors and test the robustness of input handling before deployment.

From a governance perspective, one of the biggest challenges is simply **visibility**. Many enterprises may not fully know to what extent AI is being woven into citizen-developed apps. However, surveys suggest it's already significant. Zenity's 2024 industry report found that the average large enterprise had developed over 2,600 of their own AI "copilots" (AI-driven app features or bots) using low-code platforms [46]. Worryingly, 63% of those AI components were overshared to users outside their intended team or even made accessible to the public (likely unintentionally), creating risk of prompt injection and data leakage exploits. This statistic indicates that as AI features propagate through low-code apps, improper access configurations are opening new avenues for attack. Another insight from that report was the sheer scale of the low-code/AI footprint: nearly 80,000 apps and automations in a typical large enterprise, with on the order of 50,000 collective vulnerabilities among them. These include not just AI issues but the full gamut of weaknesses, still, it underscores how introducing AI into the mix adds complexity to an already sprawling security landscape.

In summary, the combination of AI and low-code offers tremendous power to end users but expands the potential attack surface in ways that mix human and machine weaknesses. An insecure low-code app could be manipulated by an AI; conversely, an AI could be manipulated by an insecure input from a low-code app. Security teams must adapt by incorporating AI risks into their threat models and by using automated tools to keep watch. Encouragingly, awareness is growing: OWASP's inclusion of generative AI app risks and vendors launching AI-aware security solutions show that the community is beginning to tackle these hybrid threats. Still, this area will likely evolve rapidly. As organizations embrace AI-enabled low-code development, they will need to remain vigilant and update their security practices continuously, the attackers certainly will. Finally, we consider how all these issues converge into the need for governance and centralized control, and why that remains an open challenge moving forward.

4.5 Governance and Centralized Control: An Open Challenge

The rise of low-code and citizen development poses a fundamental question for organizations: how can we reap the benefits of democratized software creation without losing oversight and control? Governance in this context refers to the policies, processes, and tools that ensure applications built by anyone in the organization meet certain standards for security, compliance, and quality. Achieving effective governance over a decentralized, fast-moving low-code environment is an **open challenge**, one that many enterprises are still grappling with in 2025. The need for governance became starkly apparent after incidents like the Power Apps data leak, which demonstrated how a single misconfigured citizen app could expose millions of records. Business stakeholders and IT leaders alike recognized that without checks and balances, software democratization can lead to chaos and risk. In fact, a 2024 survey found that 47% of organizations were implementing or planning to implement formal low-code policies and governance frameworks to address this issue [47]. This represents a significant shift from just a few years prior, when low-code was often a “shadow IT” experiment. Now, nearly half of companies are actively trying to impose some centralized control, yet that also means more than half still have not or are unsure how to proceed.

Governance hurdles. What makes low-code governance difficult? First, the sheer **volume and diversity of apps** is far greater than in traditional IT. In the past, IT could maintain an inventory of all official applications (perhaps a few hundred major systems). Today, as noted, there may be tens of thousands of mini-applications floating around the enterprise [46]. Tracking them is a non-trivial task. Enterprises are investing in discovery tools and inventory catalogs for low-code assets, for example, by leveraging platform admin APIs to list all apps and flows created by users. However, having a list is only the start. Each of those apps might need classification: which handles sensitive data? Which are mission-critical vs. trivial? Which tie into regulated processes. This requires a level of **cross-functional collaboration**. Internal audit, compliance, and security teams need insight into what citizen developers are doing. One recommendation from experts is to establish a direct line from internal governance teams to all business units involved in citizen development [40]. In practice, some companies have created *Citizen Development Councils* that include representatives from IT/security and each department, to review proposals for new apps and set ground rules (e.g. “No app should store customer PII unless approved by the council”). Gartner has called for **fusion teams**, blended teams of IT and business, to supervise and support citizen development efforts [48]. The idea is to neither fully centralize (which would bottleneck innovation) nor completely decentralize (which invites incidents), but to create a guided framework.

Despite best intentions, establishing these controls is an iterative learning process. Early governance attempts sometimes face resistance from business units who fear IT oversight will slow them down. There can be a cultural clash: business users turned citizen developers often feel a sense of ownership and pride in “their” applications, and may not welcome stringent reviews or mandates from security teams. Successful governance programs thus tend to emphasize **enablement over policing**. For example, providing pre-approved templates or components that citizen developers can use to build securely (thereby channeling them into safer patterns), rather than just issuing prohibitions. Microsoft’s Power Platform provides a concept of “**managed environments**” where certain guardrails (like environment-level DLP policies) can be enforced and advanced logs collected; adopting such features can give IT more oversight without requiring constant manual intervention.

Key governance aspects. Several focal points have emerged for low-code governance. One is **access and identity management**. Many organizations are tightening how citizen developers can share their apps. For instance, enforcing that apps can only be shared internally (no anonymous public links unless explicitly approved) or integrating low-code apps with single sign-on and multi-factor authentication by default. This mitigates the risk of apps being broadly exposed. Another focus is **data governance**. Companies are defining which data sources can be used in low-code apps and which cannot. Modern low-code platforms allow admins to set data policies, for example, preventing a flow that takes data from a HR system and posts it to an external social media API. By segmenting data connectors into “safe” and “risky” categories and monitoring data flows, organizations aim to prevent accidental data leaks. **Change management** is also being reconsidered. Traditionally, any change in a production system went through change control boards. With citizen apps, changes happen informally all the time (a user tweaks their app logic on the fly). Some governance frameworks suggest introducing lightweight change logs or requiring citizen devs to document changes in a central repository, especially for widely used apps. This ties into **monitoring**: establishing monitoring on critical low-code apps nearly as rigorously as for official IT systems. If a crucial workflow built by a citizen fails or is accessed by an unusual user, someone should get an alert.

One forward-looking aspect of governance is **platform-level controls using AI**. Just as AI is a risk, it can also be part of the solution. Some vendors (like Zenity and others) are offering **AI-driven analysis of low-code environments** to flag anomalies, for example, detecting if an app suddenly starts processing far more data than before (which could indicate misuse) or if a known vulnerable pattern is present in an app's configuration. These **automated guardrails** are increasingly necessary to operate at scale. Zenity's research emphasized that current low-code adoption is *"evolving at a pace never seen before"* and lacks sufficient security guardrails and threat detection mechanisms [46]. By embedding continuous monitoring and employing machine learning to identify risky behavior across thousands of apps, organizations hope to catch issues that human governance boards might miss.

Despite these efforts, **centralized control remains elusive in many respects**. The technology and processes are still catching up to the phenomenon. It's telling that while 81% of companies say low-code is strategically important, only 31% have fully made it a central part of their development strategy (with appropriate planning and integration into IT) [46]. This gap suggests that many are in a transitional state, they know citizen development is happening and is valuable, but they haven't fully adapted their organizational controls to it. In some ways, it mirrors the early days of cloud computing, when business units adopted cloud services faster than central IT could govern them, leading to a flurry of cloud governance initiatives a few years later. Low-code is now at that inflection point.

In conclusion, governing low-code and no-code development is an open challenge that organizations must meet head-on. It requires rethinking traditional IT governance for a democratized context: establishing clarity on what business users can and cannot do, providing them with secure tools and templates, and implementing oversight mechanisms that don't stifle innovation. Those companies that find the right balance stand to turn citizen development into a secure backbone of enterprise innovation, a competitive advantage with managed risk. Those that fail to put governance around software democratization, however, risk a sprawling, unmanaged IT landscape rife with vulnerabilities. The coming years will likely see further maturation of low-code governance models, much like we saw for cloud governance. As one analyst succinctly put it, democratization of development "requires checks and balances" to succeed long-term. How effectively this open challenge is addressed will determine whether low-code fulfills its promise as a boon to digital transformation or becomes a breeding ground for unmanaged risk.

Chapter 5

The Innovation Code Framework: Improving Software Quality and Reducing Technical Debt

The previous chapter examined how the democratization of software development through AI-assisted and low-code platforms has expanded both innovation and vulnerability. As organizations increasingly rely on these accelerated methods, the boundaries between professional and citizen development have blurred, amplifying challenges of control, quality assurance, and long-term maintainability. This evolution highlights a growing dilemma: how can software ecosystems maintain reliability, security, and consistency in an era defined by speed and automation?

Chapter 5 delves into this question by introducing the *Innovation Code* framework, a pioneering initiative born from the convergence of industrial innovation and software governance. Conceived under the innovation branch of Confindustria Romagna, *Innovation Code* emerged from a concrete need to tackle one of the most persistent issues in modern software engineering: the accumulation of technical debt. Its founders recognized that the same forces driving digital transformation, automation, AI, and low-code platforms, were also accelerating the production of fragmented, unstandardized, and often insecure code. The framework was thus designed not only to make software creation faster and more accessible but to embed quality, reusability, and traceability directly into the development process.

Unlike many initiatives focused solely on enabling citizen developers, *Innovation Code* reinterprets the low-code paradigm as a professional instrument. The platform demonstrates how low-code tools, when combined with rigorous certification processes and shared governance, can serve as accelerators for software houses and enterprise IT teams. By standardizing reusable components, enforcing code certification pipelines, and automating quality controls through continuous integration mechanisms, the framework allows professional developers to deliver robust solutions at unprecedented speed without sacrificing maintainability. In this sense, *Innovation Code* transforms low-code from a mere productivity tool into a strategic infrastructure for quality assurance and compliance.

Ultimately, this chapter argues that *Innovation Code* represents more than a digital ecosystem, it is a governance model for sustainable innovation. By integrating certification, reuse, and shared responsibility, it provides a blueprint for reconciling the agility of low-code development with the rigor of secure software engineering. Through this lens, Chapter 5 explores how structured collaboration and automation can transform not only how software is written but how quality itself becomes a measurable, enforceable property of the digital transformation process.

5.1 Technical Debt as a Systemic Risk in the Age of Automation

Before introducing the Innovation Code framework, it is necessary to first examine the concept of *technical debt*, as it represents the core problem the initiative was designed to address. In an era of AI-assisted and low-code development, where speed often takes precedence over structure, technical debt has evolved from a mere development concern into a systemic risk affecting software quality, security, and long-term sustainability.

Technical debt refers to the accumulations of suboptimal, quick-fix solutions in software that, while expedient in the short term, incur a “debt” of increased complexity and future rework. In an age of high automation and rapid development cycles, this concept has transcended a mere coding metaphor and become a systemic risk for organizations. As earlier chapters discussed, modern development paradigms, including AI-assisted coding and low-code platforms, enable unprecedented speed and automation in software delivery. However, these same paradigms can amplify technical debt by enabling the proliferation of code (often generated or assembled without rigorous oversight) that might solve immediate problems but introduces hidden deficiencies [49]. Technical debt today is therefore not just a matter of individual code quality; it represents **“a ticking time bomb”** in cybersecurity and operational resilience [49].

One reason technical debt poses systemic risk is its cumulative effect on **security vulnerabilities**. Over time, shortcuts such as using outdated libraries, hard-coded secrets or ignoring proper error handling accumulate into a **“tangled web of outdated systems and patchwork solutions”** [49]. In highly automated environments, these hidden issues can propagate rapidly across integrated systems. Notably, a large portion of known cyber incidents have been traced to unaddressed technical debt. For example, unpatched or end-of-life components (a classic form of technical debt) are often **“unpatchable… a ticking time bomb of cyber risk”**, directly exploited in major attacks [50]. The 2017 *WannaCry* ransomware outbreak is illustrative: an estimated 98% of affected systems ran an unsupported OS, highlighting how legacy technology debt can open the door to systemic failure. [52]. In this sense, technical debt not only undermines maintainability but also enlarges the **attack surface** of organizations.

Furthermore, technical debt undermines **agility and reliability** in automated pipelines. Continuous integration/continuous deployment (CI/CD) systems and AI-based code generation tools relentlessly push new code into production. If that code carries unresolved debt (e.g. quick fixes that bypass tests or non-standard implementations), organizations may face brittle systems that **scale up flaws as quickly as features.** The risk is systemic because failures in one component can cascade in tightly coupled, automated systems. Industry experts warn that ignoring these debts "leaves open misconfigurations and security gaps", whereas addressing them is key to **fortifying the software supply chain** [55]. Indeed, consolidating and refactoring technical debt improves the overall security posture by reducing unnecessary complexity and eliminating duplicated or shadow systems [50]. This proactive approach also aligns with compliance needs: many emerging regulations demand up-to-date and secure software practices, and **simplifying legacy debt makes it easier to meet standards and avoid penalties** [50].

The Innovation Code framework directly addresses these challenges by embedding preventive mechanisms against the accumulation of technical debt within its development model. By enforcing standardized workflows, automated quality controls, and certified reusable components, the framework ensures that each software artifact adheres to consistent quality and security benchmarks before release. Through its integrated CI/CD pipelines based on GitHub Actions, every contribution is automatically analyzed, tested, and scored for compliance, preventing the introduction of fragile or redundant code. This systematic enforcement of standards transforms the management of technical debt from a reactive effort into a built-in feature of the development process. Moreover, Innovation Code promotes code reuse through its certified component marketplace, reducing redundancy and eliminating the tendency to "reinvent the wheel", a common source of hidden debt. By coupling automation with rigorous governance and shared accountability among coordinators and developers, the framework effectively turns debt control into a community-driven discipline. In doing so, it demonstrates how structured collaboration and certification can preserve agility while ensuring that automation does not come at the cost of long-term maintainability.

This logic naturally leads to one of the framework's most distinctive elements: its emphasis on **certified components and software standardization** as tools to institutionalize quality and prevent the reemergence of technical debt. If the previous section outlined why managing technical debt is critical, the next explores *how* organizations can achieve this in practice, through systematic reuse, certification, and the enforcement of common standards across all development activities.

5.2 Certified Components and Software Standardization

One effective strategy to mitigate technical debt and improve software quality is the adoption of **certified components** and rigorous software **standardization**. This approach involves reusing software modules that meet defined quality and security criteria, and enforcing uniform development standards across projects. By relying on certified, well-vetted components, organizations can avoid “reinventing the wheel” with quick bespoke solutions, a practice that often introduces new technical debt. Instead, they build on a foundation of trusted code. Standardization further ensures that all development follows consistent patterns, making systems more maintainable and secure by design.

The concept of **certified components** has gained traction in both industry and policy circles. In the European Union, for example, the Cybersecurity Act (Regulation (EU) 2019/881) laid the groundwork for an EU-wide **cybersecurity certification framework** covering ICT products, *software and components*. The goal is to harmonize security assurance levels across the Union, so that vendors and users can easily determine a component’s security posture [52]. Under this framework, products or software components are formally evaluated by accredited bodies against known standards, and issued certificates attesting to their security level [52]. Such schemes (e.g. the EU Common Criteria, based scheme for ICT products) play a key role in increasing trust: a **certified component** comes with an independent guarantee of conformity to security best practices [52]. In practice, using certified libraries or modules can significantly reduce both the likelihood of vulnerabilities and the effort needed to demonstrate compliance with regulations.

Parallel to formal certification, the software community has developed its own standards for component security. The OWASP *Software Component Verification Standard* (SCVS) exemplifies a community-driven framework to **“identify and reduce risk in a software supply chain”**, providing a structured set of activities and controls for managing third-party components [51]. SCVS emphasizes measures like maintaining an accurate **Software Bill of Materials (SBOM)** for each application and performing component analyses, which together improve transparency and allow teams to detect known-vulnerable dependencies early.

An **SBOM (Software Bill of Materials)** is essentially an inventory or “ingredient list” of all software components that make up an application, including libraries, dependencies, and their versions. Much like a food label lists every ingredient in a product, an SBOM provides visibility into what software elements are included, where they come from, and their potential vulnerabilities. This transparency enables organizations to quickly assess exposure when a vulnerability is discovered in a component, rather than searching blindly across systems. SBOMs are increasingly recognized as critical tools for managing software supply chain security, especially as regulatory frameworks (such as the EU Cyber Resilience Act) begin to require them. They also play a preventive role: by maintaining an up-to-date SBOM, teams can ensure that outdated or unverified components are replaced before they accumulate into technical debt.

As OWASP notes, **managing supply chain risk reduces the system's vulnerable surface area and makes technical debt more measurable as a barrier to remediation** [51]. In essence, an SBOM and standardized component vetting process help developers systematically avoid adding new technical debt in the form of insecure libraries. This approach is increasingly critical given the popularity of open-source packages; without standards, projects may inadvertently include components with unpatched flaws or incompatible licenses, incurring future "debt" when these issues must be fixed under duress.

Standardization in software development goes beyond components to encompass processes and tools. Organizations are instituting **uniform coding guidelines, testing protocols, and documentation standards** to ensure consistency. For instance, using a common framework or architectural pattern across teams can greatly ease maintenance and reduce errors, developers can more readily understand and update code that follows familiar conventions. Such standardization also often involves automation: incorporating **automated code quality checks, linters, and security scans** into the development pipeline to enforce standards continuously. Leading practices like these are reflected in secure development frameworks (e.g. NIST's Secure Software Development Framework and OWASP's SAMM), which encourage organizations to bake quality and security checks into every stage of the lifecycle. By doing so, deviations (potential technical debt) are caught and corrected early.

It should be noted that AI-assisted coding and low-code platforms, discussed in earlier chapters as double-edged swords for software quality, can actually benefit from a certified component approach. **Low-code development inherently relies on reusable components and modules**, so ensuring those building blocks are standardized and security-reviewed mitigates the risk of low-code "shadow IT" sprawl. Likewise, AI code generators can be guided to use only approved libraries or patterns. In both cases, an ecosystem of certified, well-documented components provides guardrails that contain the spread of technical debt despite rapid development. A recent European industry report stressed that **out-of-date components and ad-hoc solutions have left a large amount of technical debt, and that incentivizing the use of secure, up-to-date software is more important than ever** [53]. Embracing certified components and rigorous standardization is thus a proactive response: it injects quality at the source, preventing the kinds of hidden flaws that accumulate into systemic issues.

In summary, **software standardization and the use of certified components offer a path to higher quality and security.** They reduce variability in how software is built and ensure that what *is* built rests on trusted foundations. The next section will examine a concrete application of these principles, the *Innovation Code* initiative, which operationalizes reuse and certification in a national context, blending technical governance with collaborative development.

5.3 The Innovation Code Initiative: From Reuse to Governance

An illustrative case of blending code reuse, software certification, and collaborative governance is the **Innovation Code** project, a national initiative aimed at revolutionizing software development practices in Italy's digital transformation. Innovation Code was launched by the innovation arm of *Confindustria Romagna (Meta)* with the mission of accelerating enterprise software development (especially for small and medium-sized businesses) through **low-code techniques and a community-driven ecosystem**. At its core, Innovation Code is a **community of developers, companies, and IT professionals** dedicated to making software development more accessible, rapid, and secure. The initiative provides a structured platform where participants can develop, share, and reuse software components, under a unified set of rules and technological processes that ensure each component is **thoroughly vetted and certified** before wider use.

Reuse and Marketplace Model: Innovation Code introduces a centralized *marketplace* for software components, which is a key instrument to encourage reuse. Developers in the community can build modular software artifacts (prefabricated components) and, after following the required quality process, **publish them on the Innovation Code marketplace**. The marketplace (accessible via an online portal) serves as a repository of certified, reusable components that other members or client companies can browse and acquire for their own needs. This enables a form of *monetization* and incentive: **developers can sell certified software components to enterprises**, while enterprises benefit by drastically reducing development time and cost through ready-to-use modules. Crucially, every component in the marketplace comes with a guarantee of quality and security backed by the community's certification process. This model exemplifies how reuse and sharing, if properly governed, can create a win-win scenario: faster delivery of solutions for businesses and new business opportunities for developers, all underpinned by trust in the components being exchanged.

Technical Governance and Certification: The integrity of Innovation Code's marketplace is maintained by an elaborate governance framework codified in its Technological Regulation. All contributors must adhere to a strict workflow that embeds quality controls at every step. Key technical features of this framework include:

1. **Private Repository & Access Control:** Each software project (or component) is developed in a dedicated GitHub repository within the community's organization. These repositories are **closed (private) and accessible only to authorized community members**. This ensures that code is not publicly released until it has passed all checks, preserving privacy and control. To initiate a new project repository, an author (developer) must request permission from community coordinators, who oversee the creation and administration of that repo. Every repository is maintained by senior members to enforce security and proper management of the codebase.

2. **Automated Pipeline with GitHub Actions**: Innovation Code mandates an automated CI/CD pipeline for each repository, leveraging GitHub Actions to run a suite of checks on every code commit and pull request. These **mandatory controls** include:

    a. *Code Quality Analysis:* Automated linters and static analysis tools to ensure coding standards and detect errors or code smells.

    b. *Documentation Verification:* Checks that code is appropriately commented and documented, with clear descriptions for functions and modules.

    c. *SBOM Generation:* An automatic **Software Bill of Materials (SBOM)** is produced for each build, enumerating all dependencies (open-source libraries, modules and their versions). This transparency is *"essential for security and risk management"*, as it allows the community to track and later swiftly address any known vulnerabilities in included components.

   d. *Digital Signature (Sigstore):* Every artifact (build output) is **digitally signed by its contributors**. This provides a **guarantee of origin** assuring consumers of a component that it is authentic and hasn't been tampered with, and that authorship is traceable. Innovation Code leverages *Sigstore* (an open source signing and provenance tracking service) to facilitate this process.

   e. *SLSA Compliance:* Each project must comply with a specified level of **Supply Chain Levels for Software Artifacts (SLSA)**. SLSA is a framework of security best practices for software supply chains; by enforcing SLSA level requirements, the community ensures the integrity of build and deployment processes (for example, using verifiable builds, preventing unauthorized modifications, etc.). This is a forward-leaning measure, born from industry lessons on supply chain attacks.

   f. *Build & Test Automation:* The pipeline automatically compiles the code and runs test suites to verify that each commit integrates successfully and the software remains stable. No code can be merged that fails to build or that breaks specified tests, which guards against the introduction of unstable features.

3. **Compliance Scoring System:** A novel aspect of Innovation Code is its use of an **automated scoring mechanism** to evaluate each contribution. The GitHub Actions checks collectively produce a **score from 0 to 5** for every code change submitted. This *conformity score* reflects how well the changes adhere to the community's standards and pass all verification steps. The score is not merely for feedback; it is used as a **gate for accepting pull requests**. In practice, if a proposed change does not meet the minimum score threshold (for instance, due to insufficient documentation or a security test failure), it will not be merged until improved. By using a quantified score to enforce quality, the community creates an objective and transparent criterion for certification. Only code that achieves a passing score and is approved by the project coordinators can be merged and considered "certified compliant". This ensures that every artifact reaching the marketplace has gone through rigorous quality control.

4. **Coordinator Oversight and Reviews:** In addition to automated checks, human oversight is integral. Each repository has designated **coordinators (senior members)** who review contributions, manage issues, and have final say on approving pull requests. This dual control (automated scoring *plus* coordinator approval) provides a belt-and-suspenders governance model. The **guarantee of quality** is thereby “shared, controlled and reviewed by the authors and coordinators” of the community. Coordinators also help maintain consistency across projects and ensure that community rules (technical and behavioral) are observed by all contributors.

Once a piece of software has passed all these steps and is merged, it is considered an artifact ready for release. At this juncture, the author (with coordinator guidance) decides how to classify the artifact: either as a **reusable component** or as a full **product**.

- If it is a **Component**, it gets **published in the marketplace** of components, making it available for other community members or client organizations to download and integrate. The marketplace listing signals that the component is certified and can be reused confidently.

- If it is a **Product** (a complete application tailored to a specific end-user need), the distribution is handled differently: the final software can be deployed to a **verified cloud environment** of the client's choosing (the community currently supports certain qualified cloud providers like AWS Elastic Beanstalk). Even in this case, the community provides **automated deployment pipelines** and validation for those environments, ensuring that the release is carried out securely and in a standardized way. Thus, whether as marketplace components or deployed products, the output of Innovation Code’s process is delivered in a controlled, quality-assured manner.

Unified Governance and Collaboration: Innovation Code operates under a unified governance system that covers not only technical rules but also community conduct and project management. A *Regolamento di Condotta* (Code of Conduct) sets expectations for contributor behavior, promoting respect, clear communication, open collaboration, and constructive feedback among participants. This cultural framework is important for a sustainable collaborative ecosystem, especially since Innovation Code brings together diverse stakeholders (from freelance developers to enterprise IT consultants). The initiative leverages modern collaboration tools (for example, a community Discord server is used for communication and knowledge sharing) to build an active network of experts. By connecting professionals in a shared environment with common standards, the community fosters collective problem-solving. Indeed, one of the benefits highlighted for *beneficiaries* (clients) is access to *“a broad network of experts”* who can collaborate to tackle complex problems more effectively than any single vendor could.

From a governance perspective, Innovation Code exemplifies **shared responsibility and transparency**. Every software artifact is subject to the community’s scrutiny and every member has a role in upholding quality standards. The documentation emphasizes that the community adopts *“uniform development practices”* including standardized code management, testing, and documentation, to ensure consistent quality across all projects. It also states that *“each component... must meet high quality standards, ensuring that it is reliable, secure, and compatible”* and that all components used are *“certified by the community to guarantee their quality and security”.* In other words, the community itself acts as a certifying body, and this trust mark is backed by the rigorous processes described above.

Blending Low-Code and Traditional Development: It is noteworthy that Innovation Code focuses on low-code as a domain, aiming to harness its advantages while mitigating its risks through governance. Low-code platforms enable faster development by using visual interfaces and pre-built modules; however, as discussed in Chapter 4, they also introduce security and quality challenges (e.g. hidden code generation, lack of developer expertise in security). Innovation Code’s approach, requiring even low-code modules to undergo the same strict verification (SBOM, security tests, etc.) creates a bridge between the agility of low-code and the discipline of traditional software engineering. By doing so, it offers a path to make software development more accessible, rapid, and sustainable without sacrificing quality. The initiative thereby addresses the risks of software democratization (uncontrolled proliferation of applications) by introducing a **collaborative governance layer**. All code, whether hand-written or low-code generated, is funneled through a common pipeline of checks and community review. This ensures that even citizen-developed solutions or AI-assisted code can meet professional standards before reaching production.

In summary, *Innovation Code* stands out as a holistic model that **integrates reuse, certification, and governance**. It establishes a secure, shared repository of knowledge and components (much like an “app store” for vetted software modules) and couples it with a governance process that guarantees each contribution’s integrity. The project illustrates how a national or industry-wide effort can tackle the twin goals of innovation and security: companies gain speed and cost savings by reusing certified solutions, developers gain a marketplace and clear guidelines to create high-quality software, and the overall ecosystem benefits from elevated trust and reduced technical debt. By blending collaborative development with strong technical oversight, Innovation Code points toward the creation of a **secure and collaborative digital ecosystem**, a theme which we explore further in the next sections.

5.4 Regulatory Alignment: NIS2 and Shared Responsibility

The increasing complexity of software systems and supply chains has prompted regulators to raise the bar for cybersecurity governance. A prime example is the European Union's **NIS2 Directive** (Directive (EU) 2022/2555), which came into effect in 2024 as an update to the EU's Network and Information Security rules. NIS2 establishes a unified, stringent cybersecurity framework across member states, covering a broad range of critical sectors (energy, transport, healthcare, digital infrastructure, public administration, and more) [53]. It mandates that medium and large organizations in these sectors adopt risk management measures, report incidents, and address supply chain security, with significant penalties for non-compliance [53]. The directive's implementation embodies a philosophy of **"shared responsibility"** in cybersecurity, both within organizations (across management and IT roles) and across the EU (through harmonized standards and cooperation).

One notable innovation of NIS2 is the explicit **assignment of accountability to top management** for cybersecurity outcomes. Whereas traditionally cybersecurity was often delegated to IT departments, NIS2 *"changes the game for leadership"* by requiring management bodies to actively approve and oversee cybersecurity risk measures [54]. Executives and boards can no longer claim ignorance: the directive calls for management-level training to ensure understanding of cyber risks, and even allows authorities to hold individual managers **personally liable** for serious cybersecurity failings in their organization [54]. In practice, this means that if a major security incident occurs and is attributed to negligence (e.g. known security gaps were left unaddressed), company directors could face sanctions. The intent is to **"emphasize shared responsibility and reduce pressure on IT"** by making cybersecurity a boardroom issue, not just an IT issue [54]. As part of enforcement, NIS2 empowers regulators to impose measures like public disclosure of violations, "naming and shaming" of responsible persons, and even temporary bans on holding management positions in severe cases [54]. This top-down accountability ensures that adequate resources and attention are given to cybersecurity, aligning corporate governance with the technical realities discussed in previous sections (such as the need to manage technical debt and maintain secure development practices).

Another critical aspect of NIS2 is its focus on **supply chain security and ecosystem-wide cooperation**. The directive recognizes that an organization's security is only as strong as that of its suppliers and software components. Thus, NIS2 requires organizations to **address cybersecurity in their supply chain and supplier relationships** as part of risk management [53]. This includes vetting the security of third-party software and services, and possibly ensuring that suppliers follow secure development practices (for instance, using standards like ISO 27001 or maintaining SBOMs for the products they deliver). In effect, the regulatory burden is shared: suppliers must implement stronger security controls, and client organizations must perform due diligence. This dynamic echoes the **shared responsibility model** known in cloud computing (where the cloud provider and user each have security duties), but extends it broadly to all digital supply chains. Furthermore, NIS2's cross-sector approach encourages **information sharing and collective defense**. The idea is to foster cross-organization collaboration, where lessons and threat intelligence are shared in trust, acknowledging that cyber resilience is a common goal that transcends individual entities.

When aligning initiatives like *Innovation Code* with NIS2, we can see complementary goals. Innovation Code's emphasis on certified components and standardized security checks directly addresses supply chain concerns: if widely adopted, a marketplace of vetted components could help organizations fulfill NIS2's requirement to use secure ICT products. Moreover, the community governance model, involving both technical contributors and oversight roles, mirrors the NIS2 ethos of shared responsibility within organizations. Not only are developers and coordinators in Innovation Code jointly responsible for quality (as discussed in section 5.3), but the project itself was spearheaded by an industry association (Meta/Confindustria) in collaboration with companies, reflecting a **public-private partnership** approach. Such collaborations are strongly encouraged by NIS2 and other EU policies as a way to uplift cybersecurity maturity [53].

Yet, it is also evident that **regulatory compliance alone is not a panacea for technical debt or security gaps**. Experts caution that Europe's organizations have accumulated a backlog of security improvements (a form of technical debt) that **"will not be resolved by becoming NIS2 compliant"** on paper [53]. Simply meeting the minimum compliance requirements may not eliminate deeper systemic issues. The NIS2 directive provides a necessary baseline and accountability framework, but it also implicitly calls for greater investment in sustainable security practices, for instance, modernizing legacy systems (to pay down technical debt) and adopting advanced tools and standards proactively [53]. In this regard, initiatives like Innovation Code can be seen as answering that call: they go beyond compliance, creating mechanisms (technical and social) to ensure software is continuously developed and maintained at a high security level. By instituting continuous monitoring and updates for artifacts (the Innovation Code process includes ongoing **“monitoring and updates”** to promptly fix any newly discovered vulnerabilities in released components), the community embodies the kind of **“continuous improvement”** mindset that regulators hope organizations will adopt, rather than a checkbox mentality.

In conclusion, alignment with NIS2 involves both meeting its explicit requirements and embracing its spirit of shared responsibility. Organizations should integrate secure development frameworks, ensure management is engaged in cyber risk governance, and collaborate across the supply chain. The outcome is a heightened state of security readiness that not only avoids legal penalties but truly reduces risk. As we move forward, one can foresee regulatory and industry initiatives converging: for example, if a platform like Innovation Code becomes widespread, regulators might recognize or even endorse certified community components as meeting certain compliance needs, thereby streamlining the path to adherence with directives like NIS2. The final section looks ahead at how these threads, technical debt reduction, component certification, and regulatory compliance, weave together towards building a secure and collaborative digital ecosystem.

5.5 Toward a Secure and Collaborative Digital Ecosystem

The trends and practices discussed in this chapter point toward a paradigm shift in how software is developed and maintained, one that is **secure by design, collaborative in execution, and adaptive to change**. In the age of digital transformation, where AI and low-code are making software creation more democratized, the only sustainable way to reap their benefits (speed, innovation, accessibility) without incurring unacceptable risk is to embed security and quality considerations into the very fabric of the ecosystem. This means treating issues like technical debt, software component quality, and compliance not as afterthoughts, but as shared responsibilities across the community of stakeholders.

A **secure and collaborative digital ecosystem** would have several defining characteristics. First, **organizations and developers openly share and reuse vetted solutions** rather than building in silos. This reduces duplication of effort and enables collective hardening of common components, a bug found and fixed in a shared component benefits all users of that component. The Innovation Code framework exemplifies how such reuse can be governed so that sharing does not equate to insecurity; on the contrary, communal oversight can produce components that are more robust than any one organization might develop alone. This collaborative development of a “knowledge commons” in software aligns with open-source principles, but with additional layers of assurance (certifications, automated checks) suited for critical applications. As more entities participate, a network effect emerges: the ecosystem’s baseline quality improves, and the **cost of building secure software is amortized across the community**.

Second, this ecosystem would embrace **continuous improvement and monitoring** as a norm. Borrowing from both agile and DevSecOps philosophies, security and quality are not one-time checkpoints but ongoing processes. Technical debt is continuously identified, monitored, and addressed before it can put systems at risk.. Mechanisms like the scoring system of Innovation Code or similar metrics could be used broadly to keep track of the “health” of software assets. Participants in the ecosystem are expected to contribute back, for example, if a company using a community component discovers a new vulnerability, it feeds that information back to the maintainers so that a patch is issued for all users. This communal approach to maintenance embodies the **shared responsibility** model on a larger scale: just as NIS2 urges internal stakeholders to share responsibility, a collaborative ecosystem urges all players (suppliers, users, regulators) to share the burden of keeping the digital infrastructure secure.

Third, the future ecosystem is likely to be underpinned by **open standards and compliance frameworks** that ensure interoperability and trust. When multiple organizations are co-developing and exchanging software, having common standards (for data formats, security protocols, identity and signing, etc.) is vital. Standards bodies and policymakers can facilitate this by providing clear guidelines and certification pathways (such as the EU certification schemes). We are already seeing movement in this direction: the push for SBOMs as a standard artifact for software, the adoption of protocols like OAuth/OIDC for identity federation, and the development of assurance levels (SLSA, Common Criteria, etc.) all contribute to a lingua franca of security. A collaborative ecosystem would take advantage of these, integrating them into platforms so that participants “plug in” and automatically comply with best practices. This reduces friction in cooperation, companies can trust each other’s outputs if they know they adhere to the same security framework.

Finally, a secure collaborative ecosystem nurtures a culture of **education and responsible innovation**. As AI coding assistants and low-code platforms generate code, developers (including citizen developers) must be educated on secure coding practices and the implications of their choices. The ecosystem would provide not just tools but also knowledge, mentorship by experts, libraries of secure design patterns, and forums to discuss emerging threats (for instance, new AI-specific vulnerabilities as highlighted in Chapter 3). The aim is to create a virtuous cycle: the easier and safer it is to produce quality software, the more individuals and organizations will contribute positively, which in turn enlarges the pool of shared secure components and expertise.

In conclusion, the evolution of coding in the digital transformation era is poised to be defined by **collaborative governance and shared trust mechanisms**. The convergence of factors, from managing technical debt systematically, to certifying software components, aligning with forward-looking regulations, and leveraging initiatives like Innovation Code, sketches out a roadmap for achieving a resilient digital ecosystem. Each piece reinforces the others: for example, standardized, certified components make it easier for organizations to comply with regulations and to integrate security into AI/low-code development; strong governance and regulatory frameworks, in turn, incentivize the use of such components and the paying down of technical debt. The result is an ecosystem where security is not a barrier to innovation but a foundation for it. In this ecosystem, stakeholders collectively ensure that the software powering our societies is **secure, reliable, and worthy of trust**, thereby unleashing the full potential of digital transformation in a responsible manner.

Chapter 6

Towards a Secure and Collaborative Ecosystem

The discussion naturally evolves from the examination of software quality, technical debt, and regulatory alignment toward a broader reflection on how these challenges can be addressed within a secure and collaborative ecosystem. The previous analysis highlighted that the spread of automation, AI-assisted development, and low-code platforms has introduced new tensions between efficiency and control, democratization and accountability. While these technologies promise to enhance productivity and foster innovation, they also risk accelerating complexity and eroding traditional mechanisms of oversight.

The following section extends this reasoning by reinterpreting these dynamics through the lens of governance and cooperation. It explores how the convergence of human and machine creativity calls for a redefinition of responsibility, transparency, and shared stewardship in software production. The focus shifts from compliance and technical assurance toward the construction of frameworks that embed ethical reflection and cross-stakeholder collaboration into the digital transformation process.

In continuity with the preceding analyses, this next phase moves from diagnosing risks to envisioning solutions. It proposes a collective approach in which organizations, developers, and policymakers align technological progress with sustainable governance, ensuring that innovation remains secure, inclusive, and anchored to human values.

6.1 Summary of Findings

The findings indicate that *who* controls and understands code is changing: human developers are increasingly in a supervisory role over machine-generated solutions, while non-experts can create software through abstracted tools. This democratization of coding brings efficiency gains, but it also redistributes responsibility in ways that challenge traditional oversight. The **systemic risks** emerging from these trends are both technical and organizational. Notably, rapid AI code generation has been observed to **amplify technical debt**, code is produced faster than it can be robustly governed or maintained. Such *“quick wins”* delivered by AI can thus mask accumulating architectural complexity and flaws. Moreover, AI models tend to replicate patterns indiscriminately: if the training data or generated logic contain biases or errors, these will propagate systematically across all instances of use. In effect, a single flaw can be multiplied at scale, a phenomenon wherein **algorithmic bias** and defects become embedded in numerous systems via the same AI-driven components. This propagation of bias through code (for instance, in how data is processed or decisions are made) raises concerns that go beyond one-off bugs, introducing *structural unfairness* or security gaps that are hard to detect when code is produced opaquely by machines. Finally, the **diminishing human accountability** in AI-augmented development is a recurring theme. As adaptive tools generate more of the code, developers may unconsciously cede decision-making to algorithms, making it unclear who should answer for mistakes. Studies have warned that the convenience of AI suggestions can erode diligent review, leading to an oversight vacuum where no individual fully “owns” the code [55]. In summary, the findings portray a double-edged sword: while AI and low-code platforms accelerate production and broaden participation, they introduce systemic risks to quality and governance. Technical debt may accumulate faster than organizations can manage, biases in data or models can quietly permeate software at scale, and the clear lines of human responsibility central to traditional software engineering become blurred. These insights call for a reevaluation of how we *govern the coding process* in an era where humans and AI share the creative role.

6.2 Recommendations

Addressing the above challenges requires a *critical framework* that balances the benefits of automation with robust controls. The recommendations herein are framed to navigate the **trade-offs between efficiency and control**, and the **ethical tension between innovation and responsibility**. Rather than prescribing simple best practices, this framework urges each stakeholder group to consciously manage the interplay of rapid digital innovation with accountability safeguards. Concretely:

- **For Organizations:** Establish governance mechanisms that harness AI and low-code tools *without* relinquishing oversight. Firms should adopt secure development lifecycle standards such as NIST’s **Secure Software Development Framework (SSDF)** [56] to embed security and quality checks into fast-paced development. This includes setting policies for code review of AI-generated components, regular audits for bias or vulnerabilities, and maintaining documentation of AI contributions for traceability. Organizations must also weigh agility against risk by instituting “guardrails”, for example, restricting generative AI use in high-stakes code unless extra validation steps are in place. Regulatory compliance is part of this control framework: upcoming requirements (e.g. the EU’s **NIS2 Directive** on cybersecurity) demand that software supply chains implement rigorous risk management and executive accountability for software quality [58]. Management should therefore champion a culture where efficiency gains do not come at the expense of internal controls. In practice, that means allocating time and budget for testing and refactoring AI-produced code, and using metrics that incentivize secure, maintainable code rather than just rapid delivery. Ultimately, organizations must create an environment where innovation thrives within clear **governance boundaries**, aligning fast low-code development with standards (for instance, ISO/IEC guidelines on software quality and risk) that ensure **systemic reliability**.

- **For Developers (and Platform Engineers):** Embrace AI assistants and low-code platforms as productivity tools, but remain *in the loop* as critical decision-makers. This entails a professional responsibility to **maintain human oversight** even when automation handles routine tasks. Developers should follow established coding and testing practices with even greater rigor, for example, using peer review and static analysis on AI-written code just as they would on human-written code. They must also actively mitigate biases and errors: when using an AI code generator, a developer should validate outputs against unbiased datasets and diverse scenarios, catching issues an algorithm might overlook. Training and awareness are key; practitioners are encouraged to stay informed on **ethical AI principles** and to use tools for fairness and security (such as bias detection libraries or adversarial testing frameworks) when integrating AI components [57]. Importantly, developers need to guard against over-reliance on automation. Continual skill development (e.g. in algorithmic thinking, threat modeling or secure coding) is recommended to avoid the *atrophy* of human expertise. By adhering to frameworks like the **NIST SSDF** and company-specific AI usage policies, software engineers and citizen developers can enjoy efficiency gains *while upholding diligence*. In effect, the recommendation is a mindset: treat AI suggestions as *assistive*, not authoritative. Developers remain accountable for the final software product and should be prepared to justify and adjust any AI-generated code as if it were their own work.

- **For Policymakers and Regulators:** Provide clear guidelines and incentives that align technological innovation with the public interest, without unduly hampering progress. Policymakers should advance legal frameworks that clarify **accountability and liability** in AI-assisted development. For example, the **EU AI Act** will require risk assessments, transparency of AI systems, and human oversight for higher-risk AI applications, steps that begin to address some governance gaps [57]. Building on such efforts, regulators ought to update intellectual property and product liability laws to cover AI-generated artifacts: Who owns a snippet of code written by an AI, and who is liable if it malfunctions? At present these questions linger in a gray zone; legislation can define default rules (e.g. treating the deploying entity as responsible by default for AI outputs) to ensure there is always a liable party [60]. Beyond hard law, soft-law instruments and standards should be promoted. Governments can endorse and help develop **industry standards (ISO/IEC, IEEE)** that incorporate AI ethics and security into software engineering practices. For instance, international standards bodies are already exploring certifications for AI transparency and bias mitigation, policymakers can support these as benchmarks for trust. Additionally, regulators should encourage knowledge-sharing across the ecosystem: a *policy of transparency* where companies report incidents involving AI-generated code (similar to breach disclosures) could help the industry learn collectively. In sum, the recommendation for policymakers is to pursue a balanced regulatory approach: set **minimum safeguards** (so that efficiency does not race ahead of safety) while still enabling research and innovation. This includes funding the creation of open frameworks and tooling for auditing AI systems, and refining laws like **software liability** and **data protection** to cover the new realities of AI-developed software. Collaboration with industry and academia in creating these rules is crucial so that governance remains practical and evolves with the technology.

These stakeholder-specific recommendations form a cohesive framework. Each group, organizations, developers, and regulators, plays a part in reconciling the push for rapid, AI-driven innovation with the need for control, fairness and reliability. The underlying principle is **"trust but verify"**: leverage the productivity of AI and low-code, but institute verifiable checks and accountability at every level. By doing so, the software ecosystem can remain both innovative and secure, avoiding the pitfalls identified in the findings.

6.3 Limitations

While this research has explored governance strategies for AI and low-code development, it is important to critically acknowledge its limitations. The fast-moving integration of AI into coding is outpacing the evolution of legal and ethical frameworks, leaving what can be described as a **vacuum of clear rules and norms** in several areas. First, there is **no settled answer on authorship and ownership** of AI-generated code. Intellectual property law traditionally hinges on human creativity; however, when an algorithm produces novel code, determining the creator is problematic. Current jurisprudence suggests that content solely created by AI cannot be copyrighted, as exemplified by the U.S. Copyright Office's stance that works lacking a human author are not eligible for protection [59]. This implies that if an AI tool writes code with minimal human input, *no one* may hold its copyright, a scenario that undermines incentives and complicates software licensing. The thesis touched on this issue, but the broader legal debate is unresolved: **Who is the "author" of AI-written software, the user, the tool's provider or neither?** Until lawmakers and courts clarify this, organizations face uncertainty in leveraging AI code (e.g. can they enforce ownership of AI-generated components?) and risk exposure if such code unknowingly copies others' copyrighted patterns. Alongside authorship, **liability** for defects or failures in AI-generated code remains ambiguous. If an autonomous coding assistant introduces a critical security flaw, it is unclear whether responsibility lies with the developer who accepted the code, the company deploying it or the AI tool's vendor. There is a *legal gray area* here [60], and current product liability regimes do not neatly address software that has no single human author. Some proposed policies (for instance, EU initiatives parallel to the AI Act) aim to assign liability to the deployer of an AI system by default, but these are still in draft. Thus, one limitation of this work is that it can only highlight these open questions, a comprehensive solution for **accountability in AI-assisted development** is beyond the current scope, reflecting a gap that future research and policy must fill.

On the **ethical front**, incorporating AI into coding brings challenges that extend beyond technical fixes. **Algorithmic bias** is a persistent concern: if the AI's training data or algorithms carry latent biases (for example, under-representing certain groups or contexts), those biases can manifest in the code's behavior, leading to software that systematically disadvantages or excludes some users. While earlier chapters discussed bias in AI models, here we note the limitation that **mitigating bias in generated code** is not straightforward. Traditional software engineering ethics would hold developers to standards of fairness and inclusivity, but when code is machine-generated, detecting subtle biases requires deliberate effort (such as bias testing or external audits) that is not yet standard practice. Furthermore, there is a risk of **skill atrophy** and overreliance on automation. As developers lean on AI for routine coding, they may lose proficiency in fundamental skills or fail to develop the deeper understanding needed to catch errors. This phenomenon, akin to the “automation paradox” noted in other industries, can reduce the very human expertise that acts as a safety net. The analyses rest on the assumption that developers can effectively oversee AI output, but in practice, that oversight may weaken over time if humans become complacent. It must be acknowledged that such recommendations (e.g. urging developers to stay vigilant and trained) face this human-factor limitation. In summary, the governance approaches proposed operate within a still-maturing legal and ethical landscape. Key issues like authorship, liability, bias, and human expertise retention are not fully resolved by existing laws. These limitations underscore the need for ongoing interdisciplinary work: legal scholars, ethicists, and technologists must continue to refine frameworks so that the **evolving role of AI in coding** does not outpace society’s capacity to guide it responsibly.

6.4 Framework Proposals for Quality and Security

A recurring theme is the inadequacy of traditional metrics in capturing the new dimensions of software quality introduced by AI and low-code development. This section contrasts **quantitative** versus **qualitative** measures and proposes an evolved framework that blends both. Historically, software engineering has gravitated toward quantitative metrics: number of defects, test coverage percentages, build frequencies, lines of code produced, and other countable indicators of productivity or reliability. These metrics remain valuable, they provide objective baselines and can be automatically tracked. For example, an organization might measure that deploying AI coding assistants increased the volume of code written per week (a quantitative uptick). However, such counts alone can be misleading. A surge in code output could correlate with higher technical debt or more duplicated code. Likewise, a decrease in reported bugs might reflect superficial testing rather than true absence of faults. In the context of AI-generated code, purely numerical metrics might even encourage undesirable practices (like valuing volume of code suggestions over thoughtful design). Therefore, **qualitative metrics** must complement the picture. Qualitative assessments include code readability, maintainability, clarity of design, and alignment with user intent, attributes that often require human judgment or higher-level analysis to evaluate. They also encompass user-centric measures such as user satisfaction, accessibility, and ethical criteria (e.g. perceived fairness of an algorithm’s outcomes). A key insight is that AI can generate code that is syntactically correct (satisfying quantitative checks) but semantically misaligned with the nuanced requirements or values expected. Hence, any robust evaluation framework should incorporate reviews and criteria that capture these subtleties. This might involve structured peer reviews for AI contributions, scoring code on maintainability or architectural consistency or assessing whether the code's results are **explainable** and **justifiable** in context.

Building on established models, an extension of standard software quality frameworks is outlined to explicitly incorporate AI-specific quality dimensions. A key reference is the ISO/IEC 25010:2023 model of software product quality, which broadens the 2011 taxonomy and identifies nine core characteristics: *Functional Suitability, Performance Efficiency, Compatibility, Interaction Capability, Reliability, Security, Maintainability, Flexibility,* and *Safety* [61]. While ISO/IEC 25010 has been foundational in guiding software quality evaluation, it was conceived before the advent of contemporary AI coding tools and thus does not explicitly cover properties like transparency or bias. The evolving consensus is that we need to *augment* such models. In fact, recent standardization efforts have started to address this gap: for example, the new **ISO/IEC 25059:2023** proposes a quality model for AI systems that extends the ISO 25010 framework with additional attributes tailored to AI [62]. These include characteristics such as *user controllability*, *transparency*, *functional adaptability*, and *ethical risk mitigation*. In practical terms, **explainability** becomes a first-class quality attribute, an AI-generated module should be accompanied by information that enables developers and stakeholders to understand its logic (or at least its decision criteria) at an appropriate level. **Intent traceability** is another emerging concept: the ability to trace an AI-produced piece of code back to the requirement or intent that prompted it, ensuring that the code indeed aligns with the intended functionality or business rule. Likewise, **fairness** or avoidance of bias is considered part of software quality for AI-driven components; a system that produces technically correct outputs which are discriminatory or inequitable is, by modern standards, a low-quality system. The proposed framework therefore suggests that organizations and standards bodies integrate these AI-specific dimensions into their quality assurance processes. This could mean updating coding guidelines to mandate that every AI-generated feature undergo an explainability review (can the results be explained in terms of input features or rules?) and a fairness check (does it behave consistently across diverse inputs and user groups?). It could also mean adapting **quantitative metrics** to new forms, for instance, measuring the percentage of decisions in a system that are *auditable* or *transparent* (a metric for transparency) or counting the number of bias incidents detected in testing (a metric for fairness). By contrasting and then uniting quantitative and qualitative measures, the proposed framework aims for a **holistic evaluation of software**. We must continue using hard metrics for efficiency and correctness, but they are augmented with qualitative judgments and new metrics that capture aspects of trustworthiness. The end goal is a revised ecosystem of metrics and standards (aligned with initiatives like ISO/IEC

25059 and similar) that can **faithfully assess the quality of software in the age of AI**, including aspects of the code that relate to human values and complex socio-technical considerations, not just technical performance.

6.5 A Shared Agenda for Responsible Innovation

In closing, this thesis advocates for a shared agenda in governing the coming era of *digital autonomy*. As coding becomes increasingly autonomous, with AI systems taking on creative and decision-making functions, the governance of this autonomy emerges as a critical societal challenge. Meeting this challenge requires that an ethic of "**responsible innovation**" be woven into the very fabric of software development and deployment. Responsible innovation is more than a buzzword; it is a procedural principle that demands foresight and inclusivity at each step of technological advancement. In essence, it means *"taking care of the future through collective stewardship of science and innovation in the present"* [64]. In the context of AI and low-code ecosystems, this translates to all stakeholders (developers, organizations, regulators, end-users) actively collaborating to ensure that the tools and code we create today do not undermine the values and safety of tomorrow. Practically, this involves maintaining a **human-in-command** approach even as we integrate automation. Human oversight is not a box-ticking exercise but an ethical necessity: it ensures that human judgment, with all its contextual understanding and moral agency, remains in the loop and can veto or adjust an AI's actions when needed [63]. This principle upholds human agency against the backdrop of ever-more capable AI systems, aligning with emerging global guidelines that emphasize human oversight as fundamental to trustworthy AI. By keeping humans *in command*, we affirm that autonomy in software (no matter how intelligent) is ultimately subordinate to human values and intentions.

A sustainable **AI ecosystem** hence requires governance structures that are both robust and adaptive. *Robust*, in that they enforce accountability, fairness, and safety systematically across all actors; *adaptive*, in that they evolve with technological advances and learn from failures. This shared agenda calls for what might be described as **sustainable AI ecosystems**, environments where innovation can continually flourish but within boundaries that protect long-term societal interests. Key elements of such an ecosystem include transparent collaboration (sharing best practices and incident learnings openly, as one company's failure with AI code can instruct many others), interdisciplinary oversight bodies (bringing together technologists, ethicists, legal experts, and user representatives to guide policy and standards), and a commitment to **"human-centered"** outcomes. The concept of sustainability here is twofold: it concerns environmental and economic sustainability (e.g. managing the resource footprint of large AI models, ensuring tools enhance rather than replace human jobs over the long run), and it concerns the **sustainability of trust**. If users, customers, and the public at large lose trust in software because it behaves opaquely or unethically, the digital ecosystem risks a crisis of legitimacy. Thus, responsible innovation becomes not only a moral stance but a prerequisite for sustainable growth and adoption of new technologies. We must strive for AI systems and low-code solutions that *empower* users while safeguarding their rights and expectations, adhering to principles of privacy, fairness, and accountability as default features.

In conclusion, the evolution of coding in the age of digital transformation is not a story of machines replacing humans, but of **re-defining collaboration** between them under new rules. The preceding chapters have shown both the immense potential of AI-assisted development and the real dangers if its adoption outpaces our governance. A secure and collaborative ecosystem is attainable if we, as a community, commit to **“responsible innovation”** as our lodestar. This means continuously aligning our technical breakthroughs with the ethical and regulatory frameworks that keep technology benevolent and inclusive. It also means embracing *human-in-command* governance, where humans remain actively involved and ultimately responsible for digital systems, no matter how autonomous those systems become. The path forward is a shared one: organizations, developers, policymakers, and users must all partake in shaping norms and standards that ensure technology serves humanity’s collective interests. By cultivating an ecosystem grounded in trust, transparency, and accountability, we can harness the benefits of AI and low-code development while steadfastly **upholding human values.** This critical synthesis of innovation and responsibility will determine whether our digital transformation truly leads to a more secure, equitable, and collaborative future for all.

Bibliography

1.2 1 McKinsey & Company. (2024). What is Digital Transformation? McKinsey Explainers. Available at: mckinsey.com

1.4 2 Berman, S. J. (2014). The Digital Transformation: Staying Competitive. ResearchGate. Available at: researchgate.net

1.5 3 American Economic Association. (2025). The Value of Software. American Economic Review. Available at: aeaweb.org

1.8 4 Fish, T. (2025). Why AI Isn't a Bubble… It's Cognitive Infrastructure. Open Governance. Available at: opengovernance.net

1.9 5 Wikipedia. (2014). Heartbleed. Available at: wikipedia.org

1.10 6 World Economic Forum. (2024). Why Strong Cybersecurity Means We Must Reduce Complexity. Available at: weforum.org

1.11 7 Center for Security and Emerging Technology (CSET). (2024). Cybersecurity Risks of AI-Generated Code. Georgetown University. Available at: cset.georgetown.edu

1.12 8 Imperva. (2024). The State of API Security in 2024. Resource Library. Available at: imperva.com

1.14 9 Accenture. (2025). State of Cybersecurity Resilience 2025. Available at: accenture.com

1.15 10 Deloitte. (2024). Understanding NIS2: The New EU Cybersecurity Directive. Consulting & Risk Blog. Available at: deloitte.com

11 Kahl, A. (2023). The Evolution of Coding in the AI Era. The Bootstrapped Founder. Available at: thebootstrappedfounder.com

12 Wilfrid, D. (2020). A Brief History of Low-Code Development Platforms. Quickbase Blog. Available at: quickbase.comquickbase.com

13 Gartner. Citizen Developer Glossary. Available at: gartner.com

14 WaveMaker, Inc. (2025). WaveMaker Platform. Available at: wavemaker.com

15 Oracle. (2025). Oracle Application Express (APEX). Available at: apex.oracle.com

16 Analytics India Magazine. (2024). The Hottest New Programming Language is English. Available at: analyticsindiamag.com

17 Pearce, H., et al. (2025). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. Communications of the ACM. Available at: cacm.acm.org

18 GitHub Blog. (2022). Quantifying GitHub Copilot's Impact on Developer Productivity. Available at: github.blog

19 Dobberstein, L. (2022). NSA Urges Orgs to Use Memory-Safe Programming Languages. The Register. Available at: theregister.com

20 Gartner. (2022). Gartner Forecasts Worldwide Low-Code Development Technologies Market to Grow 20% in 2023. Available at: www.gartner.com

21 OWASP Foundation. (2025). ML02: Data Poisoning Attack. Available at: genai.owasp.org

22 Ji, J., Jun, J., Wu, M., & Gelles, R. (2024). Cybersecurity Risks of AI-Generated Code. Center for Security and Emerging Technology (CSET) Report. Available at: cset.georgetown.edu

23 Fu, Y., Liang, P., Tahir, A., et al. (2024). Security Weaknesses of Copilot-Generated Code in GitHub. arXiv. Available at: arxiv.org

24 Karliner, Z. (2025). New Vulnerability in GitHub Copilot and Cursor: How Hackers Can Weaponize Code Agents. Pillar Security Research Blog. Available at: pillar.security

25 Mello, J. P. Jr. (2024). Generative AI Software Development Boosts Productivity and Risk. ReversingLabs Blog. Available at: reversinglabs.com

26 OWASP Foundation. (2025). LLM01: Prompt Injection. Available at: genai.owasp.org

27 Carlini, N., Paleka, D., et al. (2024). Stealing Part of a Production Language Model. Available at: not-just-memorization.github.io

28 Marks, S., Treutlein, J., Bricken, T., et al. (2025). Auditing Language Models for Hidden Objectives. Anthropic. Available at: www.anthropic.com

29 Kissflow (2025). *35 Must-Know Low-Code Statistics and Trends*. Kissflow Blog Available at: kissflow.com

30 Gartner (2022). *Press Release: Gartner Forecasts Worldwide Low-Code Development Technologies Market to Grow 20% in 2023*. Available at: gartner.com

31 Alpha Software (2023). *Pros and Cons of Low-Code/No-Code Platforms*. Alpha Software Blog. Available at: alphasoftware.com

32 G2 (2023). *32 Low-Code Development Statistics to Know*. G2.com. Available at: g2.com

33 KPMG (2024). *How Low-Code Platforms Are Driving Digital Transformation*. KPMG International Report. Available at: kpmg.com

34 OWASP (2022). *OWASP Top 10 Low-Code/No-Code Security Risks.* Available at: owasp.org

35 Shulman, A. (2024). *Low code, high stakes: Addressing SQL injection*. Help Net Security. Available at: helpnetsecurity.com

36 OWASP (2022). *OWASP Top 10 Low-Code Security Risks - Account Impersonation References*. Available at: owasp.org

37 Ikeda, S. (2021). *38 Million Records Exposed - Microsoft Power Apps Data Leak*. Available at: cpomagazine.com

38 Kissflow (2025). *The Limits of Citizen Development - Real Failures*. Available at: kissflow.com

39 Nokod Security (2023). *Webinar Recap: Navigating the Risks of Citizen Development*. Available at: nokodsecurity.com

40 TechTarget (2022). *Citizen Development.* Available at: techtarget.com

41 Shulman, A. (2025). *Why Training Won't Solve the Citizen Developer Security Problem*. Available at: technewsworld.com

42 Viljoen, A. et al. (2024). *Governing Citizen Development to Address Low-Code Platform Challenges*. Available at: aisel.aisnet.org

43 Perez, J. (2024). *Prompt Injection and Data Disclosure Top OWASP's GenAI Risks*. Available at: tenable.com

44 CrowdStrike (2023). *Secure AI: Key Risks of the New Attack Surface*. Available at: crowdstrike.com

45 CRN (2025). *Palo Alto Networks Launches Prisma AI Security*. Available at: crn.com

46 Zenity (2024). *State of Enterprise Copilots & Low-Code Development.* Available at: prnewswire.com

47 KPMG (2024). *How Low-Code Platforms Are Driving Digital Transformation*. Available at: kpmg.com

48 Gartner (2022). *Press Release: Low-Code Adoption and Hyperautomation*. Available at: gartner.com

49 Grieveson, T. (2024). *Tackling Technical Debt in Cybersecurity: A Veteran's Guide*. BitSight Blog. Available at: bitsight.com

50 McManus, C. (2025). *Understanding the Hidden Cyber Risk from Tech Debt and EoL/EoS Security Gaps*. Qualys Blog. Available at: blog.qualys.com

51 OWASP Foundation. (2022). *Software Component Verification Standard (SCVS) - Project Overview*. Available at: owasp.org

52 ENISA. (2019). *EU Cybersecurity Certification Framework (EU Cybersecurity Act - Regulation 2019/881)*. European Union Agency for Cybersecurity. Available at: enisa.europa.eu

53 Hulme, G. V. (2025). *ENISA Attempts to Move NIS2 Forward with NIS360 Findings*. Available at: nexusconnect.io

54 Jensen, K. (2024). *NIS 2: Implementing Stricter Cybersecurity Governance*. WatchGuard Blog, 15 July 2024. Available at: watchguard.com

55 GitClear. (2024). *AI Code Quality Report.* Available at: gitclear.com

56 NIST. (2022). *Secure Software Development Framework (SP 800-218), Rev. 1.* Available at: csrc.nist.gov

57 European Commission. *Artificial Intelligence (AI Act) - Article 14 on Human Oversight and Annex III on High-Risk AI Requirements.* Available at: europa.eu

58 Directive (EU) 2022/2555. *NIS2 Directive.* Available at: europa.eu

59 U.S. Copyright Office. (2023). *Policy Statement on AI and Copyright.* Available at: copyright.gov

60 Buttell, A. (2025). *Who Is Liable When AI Goes Wrong?* Communications of the ACM. Available at: *cacm.acm.org*

61 ISO/IEC. (2023). *ISO/IEC 25010:2023 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Product Quality Model.* Available at: iso.org

62 ISO/IEC. (2023). *ISO/IEC 25059:2023 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Quality Model for AI Systems.* Available at: iso.org

63 European Commission. *Ethics Guidelines for Trustworthy AI.* Available at: europa.eu

64 Owen, R., Bessant, J., & Heintz, M. (2013). *A Framework for Responsible Innovation.* Available at: sciencedirect.com