ETIS | EDUCATION & PROFESSIONAL DEVELOPMENT WHITE PAPER

Becoming a Software Engineer in the AI Era

Five Dimensions of Professional Potential—and the Opportunity to Leapfrog

William T. O'Connell, Ph.D.

Former IBM Distinguished Engineer and IBM CTO, Quality — Americas | Adjunct Professor of Computer Science, Loyola University Chicago

October 2026 | ETIS Companion White Paper | Education & Professional Development

AI can accelerate the work. It cannot substitute for the engineer you must become.

Executive perspective

Software engineering is not disappearing; its expectations are evolving. AI can produce code, tests, documentation, and multi-step workflows at remarkable speed. Yet the speed of producing an artifact is not the same as the ability to engineer a dependable system. The opportunity for prepared students is substantial: develop strong foundations, sound judgment, and evidence of sustained work before employers ask for them. Standing still is a risk; preparation creates options.

This paper asks what persuades a technical leader to hire, develop, and progressively trust a university student or emerging software engineer. It proposes five dimensions of professional potential: technical foundations, engineering judgment, learning agility, collaboration, and initiative supported by reviewable evidence. The framework reflects the author's engineering leadership and experience conducting hundreds of technical hiring interviews, including final-stage evaluations, as well as exchanges with professional peers. It is a practitioner perspective, not a statistically validated hiring rubric or an IBM-endorsed standard.

The central recommendation is to build evidence over time. A strong portfolio is not a gallery of screenshots or a pile of generated code. It is a living record of engineering decisions, iterations, contributions, reviews, negative tests, and lessons learned. An ambitious student can begin developing this record well before graduation—and can thereby demonstrate readiness that conventional coursework alone may not reveal.

Resumes communicate claims. Portfolios demonstrate engineering work. Interviews test whether you understand and can defend it.

Companion perspective: Engineering Trustworthy Agentic AI Systems explains the enterprise obligations; this paper shows how emerging engineers can develop—and demonstrate—the capabilities to meet them.

I | What has changed—and what remains essential

1. The career equation is changing

AI-assisted development can lower the effort required to implement software, but not the responsibility for approving a system as fit for its business purpose. Agentic systems add the ability to interpret goals, plan actions, call tools, and coordinate work. This raises the importance of distinguishing a plausible result from a correct business outcome. A system may execute a transaction successfully and still violate policy, privacy, contractual terms, or customer expectations [1–3].

The early-career labor market reflects economic cycles, investment, organizational strategy, geography, and other factors; AI alone does not explain every hiring decision or layoff. A hiring freeze does not always mean zero hiring: organizations may still approve critical openings under tighter scrutiny when a business need and a strong candidate justify the exception. That does not guarantee an offer. It does make demonstrated readiness consequential when opportunities are selective. Grades and credentials matter, but relying on them alone is a weak strategy during technological change.

The engineering bar is rising in system-level reasoning, not abandoning programming. Engineers must understand generated code, data, concurrency, and failure behavior while also considering architecture, integration, business rules, regulatory and contractual obligations, and customer trust. A graduate need not be a senior architect; developing architectural thinking earlier means connecting implementation choices to the consequences of a real system. These expectations also inform adjacent roles—data engineering and science, cybersecurity, platform engineering, systems engineering, quality engineering, and technical project leadership—at different depths.

2. Principles endure; patterns evolve; tools move quickly

Engineering principles are durable obligations, not technology fashions: correctness and explicit invariants; abstraction and separation of concerns; cohesion and controlled coupling; clear interfaces and ownership; least privilege and defense in depth; observability, fault isolation, recoverability, and verifiable change. Architectural patterns organize those principles for particular constraints—transactions, asynchronous messaging, event-driven workflows, control planes, and independently enforced authorization. Tools implement the choices and change quickly. The engineer must distinguish why a principle matters, where a pattern fits or fails, and whether a tool improves the outcome enough to justify its cost and risk.

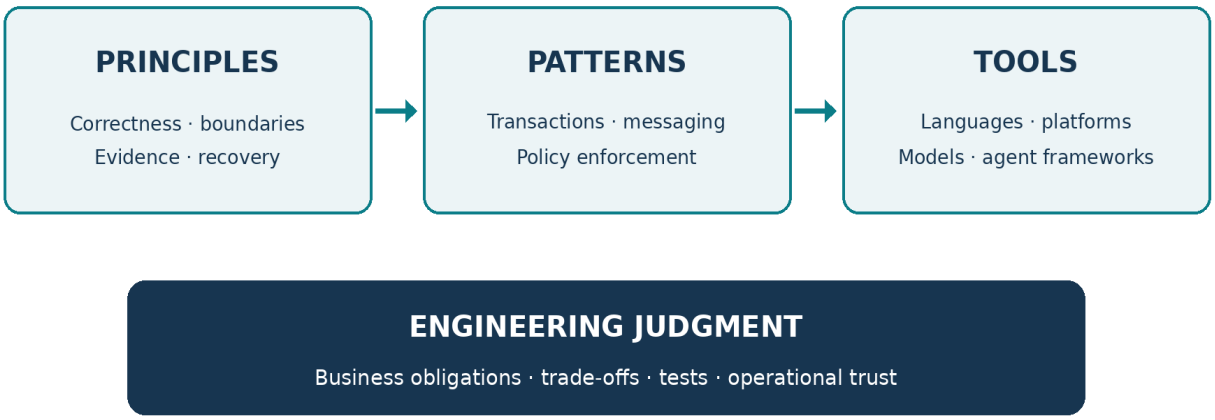

Figure 1. Principles endure; patterns evolve; tools change rapidly. Judgment connects them.

Rapidly evolving tools create a continuous learning obligation: experiment with new ways to implement, analyze, test, and verify software, but measure the benefit against correctness, lock-in, maintainability, security, and operational risk. Being comfortable with uncertainty means investigating it—not ignoring it.

The approval boundary is the practical distinction: AI proposes; an accountable engineer verifies. A happy-path demonstration may still violate a business invariant, regulatory obligation, contractual limit, privacy expectation, or customer trust. In agentic systems, governance and AI ethics must become executable architecture: authenticated identity, per-action authority, policy enforcement, traceable decisions, constrained execution, independent observation, containment, and recovery. An ethical aspiration is not an implemented control.

3. What the hiring decision actually tests

Technical leaders do not expect a new graduate to have the operational judgment of a senior architect. They do expect a foundation worth building upon. Most organizations invest in mentoring and teaching their own platforms, practices, and domain. Candidates arrive with different levels of preparation; when many compete for a selective opening, the person who can already reason through fundamentals, show learning, and substantiate their work may be the stronger investment. Hiring also depends on role fit, team needs, and circumstances; no five-part checklist guarantees an offer.

What a leader needs to know	What convincing evidence looks like
Can you reason without outsourcing judgment?	Explains assumptions, alternatives, complexity, and the test that changed a decision.
Can you be trusted with a consequential change?	Shows authorization, review, negative testing, and a rollback or recovery path.
Can you learn and accept feedback?	Shows an initial mistake, a correction, and the evidence behind the revision.
Can a team rely on you?	Shows commitments, early escalation, review participation, and shared outcomes.
Will mentoring compound your ability?	Identifies foundational strengths, gaps, and a credible plan to grow.

II | Five dimensions of professional potential

4. TECHNICAL FOUNDATIONS: know what the system is doing

The foundations include data structures and algorithms; complexity analysis; programming language semantics; software design principles and architecture patterns; databases and transactions; operating systems; networks; concurrency, parallelism, and distributed processing; testing; and security. Probability and statistics are increasingly important when software incorporates machine learning or probabilistic model output. The required depth depends on the role, but understanding must be sufficient to interrogate generated implementations and recognize failure conditions.

A student may know the time complexity of arrays and linked lists yet struggle to choose between them. That decision depends on expected operations, allocation costs, cache locality, working-set size, available memory, measured behavior, and business constraints. AI can generate technically correct code for either structure while choosing poorly if the workload and constraints are not understood. The engineer must explain which trade-offs matter, test the assumptions, and revisit the choice when conditions change. The same reasoning applies when deciding whether data should be filtered near its source or transferred to an AI service.

Governance and AI ethics become engineering work when translated into concrete requirements: who may access data, what actions require approval, which business invariants must hold, what evidence is retained, how

errors are detected, and who can intervene. Vague promises of "responsible AI" do not substitute for architecture and tests.

Demonstrate it: defend a data-structure choice against workload, memory behavior, measured performance, and business needs. Explain what AI could implement correctly yet select inappropriately—and what evidence would change your decision.

The confidence–competence gap deserves direct attention. AI can make a novice productive before they have the mental models needed to identify what they do not know. A fluent explanation of "scalability" is not enough: an interviewer may ask what happens when a message arrives twice, is delivered out of order, or times out after a transaction commits. Engineering maturity means explaining the failure, distinguishing assumptions from verified facts, seeking review, and revising the design. Confidence attracts attention; evidence earns trust.

5. ENGINEERING JUDGMENT: defend the trade-offs

There is no architecture that simultaneously maximizes every desirable attribute. Engineering judgment means framing the business problem and user needs, identifying constraints, considering alternatives, and defending trade-offs among reliability, availability, recoverability, serviceability, maintainability, performance, scalability, security, compliance, privacy, cost, and usability. The right design depends on context and evidence, not fashion.

For example, placing an authorization check in a separate trusted service may increase latency and operational complexity but provide a stronger control boundary than relying on agent instructions. A good engineer can explain why the trade-off is justified, when it may not be, and which measurements would trigger reconsideration. This is a transferable discipline, whether the system uses AI, conventional services, or a database transaction.

Probabilistic intelligence is not new to enterprise workflows. Predictive models, intelligent decision support, robotics, and automation already forced engineers to distinguish recommendations from authorized actions. Modern agents broaden the possible actions and accelerate their propagation. The established response—controlled interfaces, durable state, independent enforcement, observation, and recovery—must be reused and extended, not rediscovered [1,4].

Demonstrate it: show an architecture decision record with alternatives, rejected options, trade-offs, a negative test, and conditions for revisiting the choice.

Agentic systems make the raised bar concrete. Hostile instructions may arrive through retrieved documents or tool output; an agent may misuse delegated authority or make an apparently valid request that violates a business rule. Prompts are not a sufficient authorization boundary. Engineers must understand the components and their trust relationships, enforce authenticated identity and narrowly scoped permissions at execution, preserve authoritative state, and test both worker and supervisory failures. They must also defend the latency, availability, operational cost, and recovery trade-offs introduced by these controls. Trustworthy AI is an engineered system property, not a model characteristic.

6. LEARNING AGILITY: show how your thinking changed

Learning agility is not collecting tool names. It is entering an unfamiliar domain, forming a hypothesis, investigating evidence, asking productive questions, running experiments, and updating your understanding. It also includes learning from experienced colleagues and recognizing when a new technology is inappropriate. Curiosity without follow-through is not sufficient.

A strong candidate can describe an assumption that proved wrong, the test or review that exposed it, and the change that followed. This is often more informative than an uninterrupted success narrative. Maintain a brief learning log: initial belief, contrary evidence, experiment, revision, and next question.

Demonstrating learning agility takes more than a single example of recovering from a mistake. A sustained project over successive releases can expose the candidate to unfamiliar dependencies, feedback, revised requirements, integration failures, and technical debt. Issues, reviewed pull requests, architecture decision records, tests, and releases provide a dated record of how understanding developed. The strongest story explains what the candidate originally believed, what evidence contradicted it, who helped, and what changed in the system.

7. COLLABORATION: software engineering is a team profession

Engineering systems are produced by teams with shared interfaces, dependencies, reviews, schedules, and accountability. Communication and interpersonal effectiveness are not optional extras after technical skill. A technically capable person who consistently undermines teamwork can impose substantial costs on the entire organization. Conversely, collaboration does not mean avoiding disagreement: good engineers challenge assumptions respectfully, explain evidence, and commit to agreed decisions.

Professional behavior includes making credible commitments, reporting blockers early, reviewing peers' work, requesting help without concealment, and documenting changes that affect others. Project planning, work breakdown, issue ownership, phase gates, and continuous integration are engineering controls that expose uncertainty early. Team success does not establish equal individual competence, so contributions must be assessed through more than attendance or commit counts.

Demonstrate it: explain a dependency you surfaced early, a disagreement resolved through evidence, and a peer review that materially improved the system.

This begins in school: team projects can demonstrate planning, constructive disagreement, reviews, and dependable execution, while independent or collaborative portfolio projects provide additional opportunities for sustained ownership. A hiring conversation should probe both the technical record and interpersonal behavior: how the candidate handled a disagreement, supported a teammate, disclosed a blocker, and responded when review exposed a mistake. Team outcomes alone do not establish individual competence; neither does raw repository activity.

8. INITIATIVE: build experience before you are hired

Initiative means creating opportunities to learn and contribute within legitimate authority—not acting as a lone hero. Coursework can provide essential foundations, but a student can go further by building a substantial system, sustaining it through multiple releases, collaborating with others, and producing reviewable engineering evidence outside the minimum requirements of a class.

A student can acquire meaningful experience before a first engineering job by building and operating a system that real people use—even a small cloud service or app shared with classmates or friends. Actual use introduces changing requirements, defects, support questions, and responsibility for data and reliability. This is not equivalent to years of enterprise production accountability, but it can be persuasive evidence of preparation. The initiative is in the sustained work, not the size of the demo.

Demonstrate it: publish and maintain a useful system; preserve genuine issues, decisions, pull requests, reviews, tests, releases, and evidence of your contribution.

These five dimensions reinforce one another. Initiative creates the opportunity to build; foundations make the work intelligible; judgment shapes trade-offs; collaboration improves it; learning agility turns feedback into the next release. The living portfolio is where those capabilities become visible.

Four capabilities run across all five dimensions: business and customer understanding; security, governance, and professional responsibility; operational ownership; and epistemic discipline—the habit of separating what is

known, inferred, assumed, and verified. They are not proposed as additional dimensions. They are ways of making technical foundations, judgment, learning, collaboration, and initiative consequential in real systems.

III | The living engineering portfolio

9. Design a portfolio as an engineering program

A student or prospective-hire portfolio should tell a coherent story of sustained learning and delivery, with a version-controlled repository (for example, on GitHub) as the living record of engineering work. The strongest anchor is often one useful system developed through repeated releases over a year or more, supplemented where practical by breadth projects, open-source collaboration, and foundational problem-solving. Students can publish a small cloud application or app-store product, invite real users, and learn from operational feedback. This is a flexible strategy: course work, internships, community projects, and appropriately shareable private work can also demonstrate depth.

Portfolio component	Purpose	Evidence to preserve
Anchor system	Demonstrate sustained systems engineering	Architecture, releases, decisions, verification, operational lessons
Breadth work	Show transfer across domains and technologies	Small, explained implementations and trade-offs
Open-source or team contribution	Work within someone else's constraints	Issues, reviews, discussion, accepted changes
Foundational practice	Show algorithmic reasoning and discipline	Complexity analysis, alternatives, edge cases, explanations

Do not manufacture activity for appearances. A repository filled with copied boilerplate, synthetic issues, or decorative pull requests can undermine credibility when probed. Likewise, an inactive or private repository does not prove incompetence; access, confidentiality, and circumstances vary. The relevant question is whether a candidate can provide authentic, appropriately shareable evidence of engineering work.

Start before senior year when feasible. An anchor system can grow over one or two years, with each release adding authentic complexity and a corresponding record of decisions and verification. Students reading this late in their degree—or after graduation—should begin now rather than wait for an ideal schedule. An appropriately scoped, credible project is better than an elaborate plan that never produces evidence.

10. A release-by-release progression

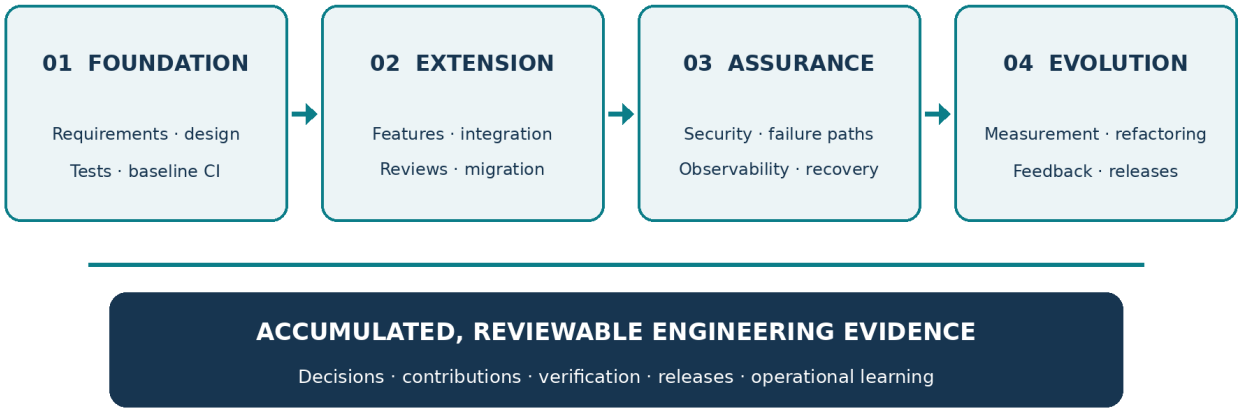

Figure 2. A living portfolio accumulates authentic engineering evidence across releases.

Cycle	Engineering emphasis	Gate evidence
Foundation	Problem definition, requirements, architecture, initial service	Scope, ADR, interface contract, tests, CI, peer review
Extension	New feature, persistence, integration, changing requirements	PR discussion, migration plan, regression tests, traceability
Assurance	Threats, negative paths, observability, failure/recovery	Authorization test, logs/traces, incident exercise, recovery proof
Evolution	Performance, maintainability, scaling, technical debt	Measurements, revised ADR, refactoring evidence, release notes

These cycles should be genuine engineering increments. Each begins with an explicit objective, records decisions, uses code review and automated verification, and closes with evidence that the increment is fit to proceed. A phase gate is not an administrative ceremony: it is a decision point where the team assesses what is known, what remains risky, and whether to proceed, proceed with conditions, or hold.

Your engineering work leaves a digital evidence trail: issues, discussions, pull requests, reviews, test results, design changes, releases, and fixes. It should be authentic, not staged for a recruiter. “Portfolio theater” means copied code, synthetic issues, decorative reviews, or superficial tests that simulate process without meaningful decisions. Reviewers may trace a claim from requirement to design, implementation, verification, and operational consequence. Missing artifacts need explanation; inflated claims undermine credibility.

11. What a reviewer can learn from your GitHub repository

An authentic repository can reveal more than the final code. Issues show problem decomposition and ownership; pull requests reveal reasoning and review behavior; commits and diffs show the evolution of implementation; CI results expose verification discipline; architecture decision records preserve trade-offs; release notes establish delivered scope; and incident or defect records demonstrate how the team learns. No single metric is decisive. A reviewer should interpret these artifacts in context and confirm the candidate’s individual contribution.

Artifact	Useful signal	What it does not prove alone
Issue / milestone	Scope, prioritization, dependency awareness	That work was completed correctly
Pull request / review	Reasoning, communication, feedback	That the author understood every

	response	dependency
Commits / diffs	Change history and ownership clues	Quality, effort, or fairness by volume
CI and tests	Repeatable checks and failure evidence	Absence of untested risk
ADR / design diagram	Trade-off reasoning and architectural intent	That implementation matches intent
Release / incident record	Delivery and learning across cycles	That the product is production ready

A serious technical interviewer may probe beneath the repository presentation. Open a specific pull request, explain what changed and why, identify reviewer feedback, and demonstrate the resulting test. This is more convincing than claiming "I built the whole application" when the evidence does not support the assertion.

A reviewer should also test foundational reasoning beyond the displayed project: algorithms and complexity, data modeling and indexes, transactions, concurrent updates, message ordering and duplicate delivery, eventual consistency, network timeouts, API design, authentication and authorization, threat modeling, and observability. The depth varies by role. The expectation is not expert mastery of every topic, but the ability to recognize the relevant ones, reason carefully, and identify where further validation is needed.

12. Example: evolving an AI-assisted customer service workflow

Consider a student team developing a simulated order-support application. The initial release implements identity, orders, and a deterministic refund eligibility service. A later release adds an AI assistant that interprets customer requests and proposes actions. Another adds supervisory monitoring, bounded retries, audit evidence, and failure recovery. The AI feature is not the entire project: it sits inside a designed business system.

A model proposes a $100,000 refund for a $100 order. A prompt may discourage it, but only an independently enforced business limit can reliably deny the request. The student should demonstrate the denial at the service boundary, verify that no transfer occurred, and test that retries do not duplicate effects. An agent saying "completed" does not prove settlement; the workflow must consult authoritative business state [1].

The engineering conversation then moves beyond the demo: Why place the rule in the trusted service? What latency or availability cost does the boundary introduce? How are policies changed? What happens when the gateway fails? What does the system log, and who may see the log? What if the supervisor is wrong? What evidence would justify releasing the next increment? These questions distinguish AI tool usage from systems engineering.

Now war-game the workflow beyond the obvious excessive-refund case. What if two agents initiate refunds for the same order concurrently? What if a payment provider accepts a refund but the network response times out? What if a retrieved customer message contains instructions to obtain elevated credentials? What if the supervisor repeats a false "completed" claim, or a policy changes while a workflow is in progress? Each case demands a different combination of isolation, idempotency, reconciliation, scoped authority, policy versioning, evidence provenance, containment, and recovery. A model API call and attractive interface alone do not demonstrate that engineering.

13. An architecture decision record (ADR): an inspectable trade-off

An architecture decision record (ADR) briefly records a consequential design choice, its context, alternatives, consequences, and evidence. The following compact example illustrates the level of reasoning a student should be able to defend.

Decision: enforce refund eligibility and amount limits in a trusted transaction service, not solely in agent instructions. Context: probabilistic output may request a business-invalid action. Alternatives: prompt-only control, validation inside the agent application, independent service authorization. Chosen approach: service-

side authorization bound to order state, policy, and identity, with audit evidence. Costs: an additional service dependency, latency, and operational overhead. Negative test: excessive refund is denied before any transaction effect. Revisit when business rules, threat model, or measured costs change.

An architecture decision record is valuable because it makes engineering judgment inspectable. It should be brief enough to maintain, but substantive enough that a reviewer can understand the decision, alternatives, consequences, and evidence.

IV | Turning the portfolio into a hiring conversation

A portfolio conversation becomes a hiring conversation when the candidate can defend the actual architecture, not merely a model wrapped in a user interface. Why use probabilistic intelligence for interpretation but deterministic services for authorization and state transitions? Which trust boundary protects business data? What happens if the gateway is unavailable? What evidence would justify release? The same record should support technical questions, collaboration stories, and independent assessment of individual contributions.

14. Two candidates, one technical discussion

Candidate A presents an attractive AI-generated application with numerous features but cannot explain the data model, trust boundaries, failures, or individual contribution. Candidate B presents a smaller system, walks through a rejected architecture, demonstrates a failed authorization test, and shows how feedback changed a release. Both may possess strengths not visible in this simplified example. Yet Candidate B offers more direct evidence of reasoning, verification, and accountable learning.

The lesson is not that appearance or productivity are irrelevant. Reviewable evidence of judgment, verification, and accountable learning can create a material advantage in a competitive hiring process, although role requirements, team fit, and the candidate's broader record still matter. An experienced interviewer can probe beyond the demonstration into fundamentals and ownership.

15. Prepare for the full range of evaluation

Hiring processes vary substantially. Depending on the role, candidates may encounter résumé screening, coding or algorithmic assessments, project walkthroughs, systems questions, behavioral interviews, and discussions with experienced engineers. Historical examples from major technology employers can illustrate the breadth of preparation, but should not be presented as a universal or current interview sequence.

Prepare for data structures and algorithms with explanation, not memorized solutions. Discuss time and space complexity, edge cases, alternatives, and why a technique applies. Prepare a five-minute architecture walkthrough and a deeper twenty-minute technical conversation. Use a concise situation–task–action–result narrative for collaboration and initiative, but substantiate the story with real decisions and outcomes.

Question to practice	What it reveals
What problem were you solving, and whose constraints mattered?	Problem framing and business context
Which design did you reject, and why?	Trade-offs and judgment
What fails if a dependency times out or a message is duplicated?	Distributed systems reasoning
What did you personally do, and what changed after review?	Ownership and collaboration
Which claim can you prove with evidence?	Verification discipline
Where did AI help, where was it wrong, and where did you avoid it?	Appropriate AI use and independence

An interview may move rapidly from a five-minute project narrative into a deep technical challenge. Candidates should be ready to draw a component diagram, explain a failure sequence, reason about a race condition or duplicated message, and defend why they selected a synchronous API, queue, transaction, or compensating workflow. Behavioral questioning is equally substantive: reviewers may test whether the candidate shares credit, accepts feedback, and communicates uncertainty without concealment. There is no universal interview funnel, but depth of understanding is difficult to counterfeit.

16. The AI distinction: accelerate development and engineer intelligence

There are two distinct ways students should understand AI. First, AI assists the engineering process: requirements analysis, coding, refactoring, tests, documentation, review, and troubleshooting. It can increase throughput, but its output still requires verification. Second, AI becomes part of the delivered system, where probabilistic reasoning, tool use, authority boundaries, workflow state, privacy, and business correctness become architecture concerns. The second requires substantially more than knowing how to call a model API [1].

A strong portfolio can demonstrate both. Show which artifacts were AI-assisted, how you verified them, and why you chose not to use AI in some decisions. Then, if your system includes intelligence, explain how you bounded its authority and verified its business consequences. Students need not build an agentic application to be strong candidates; the underlying engineering disciplines apply across systems.

V | Opportunity, apprenticeship, and the next ninety days

17. Leapfrogging without skipping experience

Major technology transitions create openings for newcomers who learn emerging capabilities early and combine them with durable foundations. A student prepared in distributed systems, security, design, verification, and governed AI may contribute in an area where an organization is still developing expertise. That can accelerate responsibility; it does not instantly confer the judgment earned through production incidents, customer commitments, and long-term ownership.

The durable strategy is to build on prior art. Enterprise systems incorporated predictive models, robotics, business process management, and intelligent automation before LLMs. The intelligence has changed dramatically; the obligation to govern its consequences has not. Experienced engineers who adapt remain essential teachers and collaborators, not obstacles to bypass.

We are living through a consequential transition. Earlier waves—including networking and client/server systems, the web, cloud, and data-intensive platforms—opened new engineering opportunities without making all prior technologies or their specialists obsolete. Prepared newcomers can sometimes move faster into emerging work, while experienced engineers who adapt provide indispensable context and mentorship. The opportunity to leapfrog is to accelerate learning and contribution—not to skip the engineering apprenticeship.

18. The experience-ladder risk is shared

Implementation, debugging, testing, maintenance, and code review have traditionally helped junior engineers develop judgment. If AI removes some of that work without replacing the learning opportunities, organizations may weaken the pipeline from junior to senior talent. The extent of this effect is not yet settled. It is a risk that universities and employers should monitor and address, not a prediction to present as inevitable.

Students should seek feedback and ownership. Universities should assess individual reasoning, collaboration, and verification alongside team results. Employers should provide mentoring, design reviews, progressively

consequential work, and operational learning. Apprenticeship is a shared obligation, not a benefit that students can manufacture alone.

The opportunity becomes actionable through deliberate practice. Identify the capabilities you need, choose meaningful work, establish milestones, and accumulate reviewable evidence. The following ninety-day program is a starting cycle for engineering your own development—not a substitute for sustained learning.

19. A practical ninety-day program

Period	Work	Reviewable result
Days 1–30	Choose an anchor problem; establish requirements, architecture, initial tests, and two alternatives; refresh relevant fundamentals.	Problem brief, diagram, first ADR, baseline CI
Days 31–60	Deliver a meaningful increment; invite review; add negative tests, a dependency-failure case, and explicit AI-use verification.	Reviewed PR, test trace, revised ADR, contribution record
Days 61–90	Release a second increment; run a recovery exercise; improve documentation; practice technical and behavioral walkthroughs.	Release evidence, recovery result, five-minute demo, next-cycle plan

Ninety days is a starting cadence, not a substitute for a sustained one- or two-year portfolio. Scope the work to available time, equipment, and access. Depth, authenticity, and learning matter more than project size.

The ninety-day plan is an entry point, not a credential. Revisit the anchor system repeatedly: measure behavior under load, introduce a dependency failure, run an access-control negative test, review an architectural alternative, and document an actual improvement. A compelling portfolio shows that engineering quality accumulates across releases—and that the candidate can distinguish a working demo from an operationally credible system.

Once a student has built credible evidence, the next question is where that potential can grow. Evaluate prospective employers not only by company reputation or compensation, but by the quality of technical feedback, mentoring, review, and progressive responsibility.

20. Choose the right employer—and engineering team

A recognizable company can provide excellent mentoring, but prestige does not guarantee the quality of a particular team. Ask for concrete examples: how a junior engineer receives design feedback, participates in code review, responds to a production problem, and earns broader responsibility. Ask who mentors new hires, how disagreements are resolved, and whether incidents lead to system improvements rather than blame. Evaluate the answers—not just the company name or recruiting presentation.

Employers will teach internal technologies and business context. Students should arrive with the foundations that make that investment worthwhile. A healthy learning environment and a prepared candidate reinforce one another.

Career preparation itself can be managed as an engineering project: identify gaps, define outcomes, choose a realistic scope, deliver evidence in iterations, invite review, measure progress, and revise the plan. The ninety-day program is an initial cycle, not a credential or a substitute for years of learning.

21. The three-party commitment

No one party can sustain this pathway alone. Students own their preparation; universities must make reasoning and individual contribution visible; employers must preserve mentoring and progressively consequential work. Many organizations already invest seriously in this apprenticeship. The challenge is to maintain and adapt it as AI changes the tasks through which junior engineers traditionally learn.

Students	Universities	Employers
Master fundamentals; practice independent reasoning; build authentic evidence.	Teach enduring principles and current systems; evaluate reasoning and technical foundations.	Hire for demonstrated potential; provide mentoring and progressive responsibility.
Contribute reliably; expose blockers early; seek and use feedback.	Assess individual and team contributions; require reviews and negative tests.	Preserve learning pathways; use code and design review to develop judgment.
Use AI transparently; defend the system's behavior and business consequences.	Teach governed AI architecture rather than only tool use.	Invest in trustworthy engineering and the next generation of technical leaders.

Conclusion | Become the engineer behind the artifact

Your degree matters. So do your technical foundations. But a degree alone cannot demonstrate how you respond when a design fails, a teammate challenges your assumption, or a system violates a business rule. Build something useful. Operate it. Improve it. Show the decisions you made, alternatives you rejected, tests that failed, reviews you accepted, teammates you helped, and evidence that supports confidence in the result.

Do not wait for your first engineering job to begin thinking and acting like an engineer. Your university education provides essential foundations; developing professional capability also requires deliberate choices beyond completing coursework. Identify the engineer you want to become. Design a learning plan, build meaningful systems, confront failures, seek demanding reviews, and revise your decisions as evidence develops. Your living GitHub portfolio is the traceable record of that journey: not just what you built, but how you learned to reason, collaborate, decide, and take responsibility. Do not claim experience you do not have; demonstrate the engineer you are becoming.

AI can accelerate the work. It cannot substitute for the engineer you must become.

Scope and companion reading

This practitioner-informed educational white paper is neither a predictive labor-market model nor a validated hiring instrument. Expectations vary by role, specialty, region, and team; examples are constructed instructional scenarios. The companion ETIS technical white paper [1] supplies the deeper architecture behind this paper's claims: governed agentic autonomy, independent trusted execution, authoritative state, intelligent supervision, evidence, containment, and recovery. Read together, the papers connect professional preparation to the engineering obligations of systems that organizations and customers can trust.

References

[1] O'Connell, W. T. (2026). Engineering Trustworthy Agentic AI Systems: Governed Autonomy, Intelligent Supervision, and the Protection of Business Trust. ETIS Technical & Strategic White Paper, Version 9.

[2] National Institute of Standards and Technology. (2023). Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1. https://doi.org/10.6028/NIST.AI.100-1

[3] National Institute of Standards and Technology. (2022). Secure Software Development Framework (SSDF), Version 1.1. NIST SP 800-218. https://doi.org/10.6028/NIST.SP.800-218

[4] IBM. The Evolution of AI Agents. https://www.ibm.com/think/topics/evolution-of-ai-agents

[5] National Association of Colleges and Employers. Career Readiness Competencies. https://www.naceweb.org/career-readiness/competencies/career-readiness-defined