

DOI: https://doi.org/10.48009/3_iis_2026_102

Automated skills orchestration: Teaching agentic AI architecture through deterministic execution frameworks

Bryan Marshall, *Georgia College & State University*, bryan.marshall@gcsu.edu

Peter Cardon, *University of Southern California*, cardon@marshall.usc.edu

Brad Fowler, *Georgia College & State University*, brad.fowler@gcsu.edu

Abstract

Agentic AI systems—autonomous workflows capable of multi-step reasoning, tool invocation, and self-correction—are reshaping how information systems professionals work. Yet a tension underlies them all: large language models are stochastic by design, producing variable outputs through temperature-controlled token sampling, while business processes demand consistency and reproducibility. This paper proposes a three-layer skills architecture for teaching IS students to navigate that tension: a Directive layer (Markdown standard operating procedures that define intent), an Orchestration layer (LLM-powered decision-making on platforms such as AWS Bedrock), and an Execution layer (deterministic Python that handles computation, API calls, and data processing). Drawing on the practitioner-developed skill pattern, we formalize the architecture as a pedagogical framework, apply it across IS domains—database management, accounting, business intelligence, and project management—and illustrate it with a skill deployed in production. The central argument is that IS programs face a consequential choice: teach students to *use* AI tools, a competency with a short half-life as tools change and tool use is itself automated, or teach them to *architect* AI systems, a competency that remains valuable regardless of which tools prevail because the architectural principles are invariant. Students who build skills through this framework are not learning to use AI tools; they are learning to design the systems that use them.

Keywords: agentic AI, skills orchestration, deterministic execution, curriculum, LLM architecture, AI pedagogy

Introduction

The interaction model between humans and artificial intelligence is undergoing a fundamental transformation. The initial wave of large language model applications—chatbots, writing assistants, code completions—established AI as a tool that users invoke for discrete tasks. The emerging wave is categorically different: agentic AI systems that operate autonomously, perceive their environment, plan multi-step actions, invoke external tools, and reason about whether outcomes match intent.

This transition from chatbot to agent introduces an architectural challenge that current IS curricula do not address. Pure LLM-based approaches produce systems that are unpredictable in ways that matter for business operations—they hallucinate facts, drift in reasoning over long contexts, and produce different outputs for identical inputs (Vaswani et al., 2017; Brown et al., 2020). The professional reality is that production agentic systems must combine the flexible reasoning of LLMs with the predictable execution of deterministic code.

The professional challenge is not simply learning to use AI but learning to architect systems that leverage its strengths while constraining its weaknesses—knowing what should be probabilistic and what must be deterministic, and designing the interfaces between them. The three-layer skills architecture presented here—Directive, Orchestration, and Execution—provides a concrete, teachable framework for that separation, drawing on practitioner methodology from production agentic systems (Saraev, 2025).

The Compounding Error Problem

The urgency of teaching deterministic separation becomes apparent in how errors compound across multi-step agent workflows. If a single LLM call has 95% reliability, a five-step workflow that relies on the LLM for every operation drops to roughly 77% end-to-end—and a ten-step workflow falls below 60%. Business workflows routinely involve dozens of operations; purely probabilistic execution does not survive the compounding.

The solution is not to make LLMs more accurate at deterministic tasks—that conflicts with their architectural properties—but to remove deterministic operations from the probabilistic path entirely. Each operation moved to deterministic code removes a probabilistic point of failure—under controlled conditions it executes reproducibly—so end-to-end reliability degrades only with the probabilistic operations that remain. (This claim is about the stochastic error introduced by the model; Section 3.5 qualifies what reproducibility does and does not guarantee in production.)

The Pedagogical Imperative: Architects, Not Operators

The current approach to AI integration in IS education focuses on tool usage—teaching students to write prompts, evaluate outputs, and integrate AI tools into workflows. This is valuable but insufficient: tool-specific knowledge has a short half-life, as both prompts and platforms are quickly supplanted. What has lasting value is architectural thinking—decomposing a problem into probabilistic and deterministic components, designing the interfaces between them, and reasoning about the reliability, maintainability, and governance of the composed system. IS programs can graduate consumers of AI products or architects of AI systems, and the architects command the wage premium.

Research Questions

This paper addresses four research questions:

1. What architectural framework can guide the design of reliable agentic AI systems that appropriately separate probabilistic reasoning from deterministic execution?
2. How can the practitioner-developed skill pattern be formalized as a pedagogical framework for IS education?
3. What role can enterprise orchestration platforms such as AWS Bedrock play in teaching students to build production-grade agentic systems?
4. How should this framework be applied across IS curriculum domains, and what does assessment look like when the learning objective is system architecture rather than tool usage?

Literature Review

The Stochastic Nature of Large Language Models

Large language models generate text through autoregressive token prediction: given a sequence of tokens, the model produces a probability distribution over the next token, samples one, appends it to the sequence, and repeats (Vaswani et al., 2017). The output is stochastic by design — identical inputs can produce different outputs.

Generation parameters (temperature, top-k, top-p) trade off determinism against creativity. Lower temperature concentrates probability mass on the most likely tokens; higher temperature spreads it across the distribution (Brown et al., 2020). Even at temperature zero, models are not fully deterministic because of floating-point non-associativity in GPU computation and tie-breaking in sampling (Fan et al., 2018; Holtzman et al., 2020). This stochasticity is a feature, not a bug — it enables the flexible reasoning LLMs provide — but it is also a constraint for business operations.

Agentic AI Systems

The evolution from single-turn prompt-response interactions to multi-step autonomous agents represents the current frontier of AI deployment. The defining capability of an agent is *tool use*: the model interleaves reasoning with actions that invoke external functions, observes the results, and revises its plan. ReAct established this reasoning-and-acting pattern, showing that interleaving chain-of-thought with tool calls improves both task performance and interpretability over reasoning or acting alone (Yao et al., 2023). Toolformer demonstrated that models can learn, in a self-supervised manner, which APIs to call, with what arguments, and how to fold the results back into generation—offloading tasks such as arithmetic and factual lookup, where smaller deterministic tools outperform the model itself (Schick et al., 2023). Reflexion added a self-correction loop in which an agent verbally reflects on feedback from a failed attempt and stores that reflection to improve subsequent trials (Shinn et al., 2023)—the scholarly antecedent of the self-annealing loop developed in Section 3. Survey work organizes these developments into a common framework for constructing and evaluating LLM-based autonomous agents (Wang et al., 2024).

A consistent architectural pattern underlies both this peer-reviewed work and production agentic frameworks (LangChain, AutoGen, AWS Bedrock Agents): an LLM serves as the decision-making core, reading instructions and invoking external tools, while the tools themselves are deterministic code (Chase, 2023; Wu et al., 2023; Amazon Web Services, 2024). What this literature establishes empirically—that capability and reliability improve when deterministic work is delegated to tools rather than performed by the model—the present paper develops into an explicit, teachable architecture.

Deterministic vs. Probabilistic Computing

The distinction between deterministic and probabilistic computing is foundational to computer science but has taken on new pedagogical urgency. Traditional software engineering trained students to write deterministic programs — given the same inputs, the same outputs. LLMs introduce a different paradigm where outputs follow a distribution.

The resolution is not to choose between paradigms but to route each operation to the computational paradigm whose properties match its reliability requirements. This is an application of a long-standing principle in software architecture: Parnas (1972) argued that systems should be decomposed into modules around the design decisions each one hides, so that components can be designed, tested, and changed independently. The three-layer architecture applies that principle to the stochastic–deterministic boundary, isolating probabilistic reasoning behind a stable interface so deterministic work can be verified on its own terms.

Current Approaches to Teaching AI in IS

Current IS curricula incorporate AI in several ways: adding AI-focused courses, integrating AI tools into existing courses, and emphasizing responsible AI use. Recent IS-education scholarship has concentrated largely on *when* generative AI use is appropriate—for example, Firth et al. (2024) offer a framework for teaching students to distinguish legitimate AI assistance from academic dishonesty. What is largely absent is the architectural perspective — how to design AI systems that are reliable, maintainable, and governable at production scale (Marshall, in press).

This gap is analogous to teaching students to use a database versus teaching them to design one. Both are valuable, but the architectural skill commands the higher wage premium and is more durable as the underlying technology evolves.

The pedagogy proposed here is grounded in established learning theory even where the specific framework is not. Building, testing, and refining a working skill is a form of experiential learning, in which understanding is constructed through a cycle of concrete experience, reflection, and active experimentation (Kolb, 1984); the self-annealing loop introduced in Section 3 operationalizes exactly this cycle. Assessing students on the artifacts they build, rather than on recall, reflects constructive alignment between intended outcomes, learning activities, and assessment (Biggs, 1996). We note candidly that the three-layer skills-orchestration framework itself originates in practitioner methodology (Saraev, 2025) and has not yet been the subject of peer-reviewed IS-education research. That absence is not a defect in the paper's grounding but the gap it sets out to address: the framework's components—agentic tool use, separation of concerns, and experiential learning—are well established in the peer-reviewed literature, while their synthesis into a teachable architecture for IS curricula remains, to our knowledge, novel.

Theoretical Framework: The Three-Layer Skills Architecture

The three-layer architecture separates concerns across three distinct layers, each with a well-defined role, a natural representation format, and a specific relationship to the stochastic-deterministic spectrum. This separation is the central contribution of the framework: it provides students with a mental model for decomposing any AI-augmented workflow into components that can be independently designed, tested, and maintained.

Layer 1: Directive (The Brain)

The Directive layer defines *what* the system should accomplish. Directives are written in Markdown — natural language augmented with structure — and reside in a SKILL.md file at the root of each skill folder. They specify the skill's purpose, the conditions under which it should be invoked, the procedure it should follow, and the references it requires.

Markdown is a deliberate choice: it is human-readable, version-controllable, and natural for LLMs to parse. Directives are not executable — they are read by the orchestrator, which then performs the actions described.

Example directive structure:

name: `query-validator`

description: >

Use this skill when a user submits a SQL query for review.

Validates query correctness, evaluates performance via EXPLAIN, and returns optimization recommendations.

Procedure

1. Receive the SQL query from the user.
2. Run ``python3 scripts/validate_syntax.py --query "{query}"`` to check syntax and ``python3 scripts/run_explain.py --query "{query}"`` to analyze the execution plan.
3. Review the EXPLAIN output for full table scans and missing indexes; provide the user a summary of findings and optimization recommendations.

Layer 2: Orchestration (The Decision-Maker)

The Orchestration layer is the domain of the large language model, where probabilistic reasoning belongs: interpreting user intent, selecting the appropriate directive, sequencing tool invocations, and making judgment calls when inputs are ambiguous or results unexpected. The orchestrator does not compute, process data, or call APIs directly; it reads directives, calls execution scripts in the right order, interprets the results, and communicates findings onward. AWS Bedrock is the recommended enterprise platform for this layer (Section 5).

Layer 3: Execution (The Hands)

The Execution layer is where deterministic computation occurs. This layer consists of Python scripts that handle operations requiring consistency, precision, and reproducibility: database queries, API calls, mathematical calculations, file transformations, data validation, and report generation. The defining property of the Execution layer is reproducibility: under controlled conditions, identical inputs produce identical outputs, making each step independently testable and auditable (a property we distinguish from correctness and from operational reliability in Section 3.5). Python is the recommended implementation language (Section 6).

Why This Separation Works

The three-layer separation addresses the compounding error problem directly. Every operation moved from the Orchestration layer to the Execution layer replaces a probabilistic step with a reproducible one, removing it as a source of the compounding *stochastic* error (subject to the operational caveats of Section 3.5). The probabilistic layer is reserved for what requires probabilistic reasoning — natural language understanding, ambiguous decision-making, creative synthesis. Everything else runs deterministically.

The separation also enables a **self-annealing loop**: when a script in the Execution layer fails, it produces a deterministic error message. The orchestrator reads the error, diagnoses the cause, and the directive can be updated with the new knowledge. Each failure makes the system stronger.

1. An error occurs in a deterministic script.
2. The orchestrator reads the error message, diagnoses the cause, and fixes the script.
3. The directive is updated with the new knowledge (e.g., “API rate limit is 100 requests/minute; batch accordingly”), leaving the system stronger than before.

This pattern mirrors professional debugging practice and provides a natural pedagogical framework for teaching diagnostic reasoning.

Determinism Is Not Correctness

A careful distinction must be drawn between *determinism* and *correctness*, and between *reproducibility* and *operational reliability*. Moving an operation from the Orchestration layer to the Execution layer makes it reproducible and testable; it does not make it correct. A script can be deterministically wrong: a logic bug, a malformed input, an off-by-one error, or a mishandled edge case will produce the same incorrect result every time. Determinism guarantees only that the behavior—right or wrong—is consistent, and therefore observable, debuggable, and fixable. This is itself valuable: a reproducible failure can be diagnosed and corrected, whereas a stochastic one is difficult to even characterize. But reproducibility is a precondition for reliability, not a substitute for it—a system can be *reliably wrong*.

Operational reliability in production depends on far more than deterministic computation. The Execution layer routinely touches conditions outside its control: external APIs that change or fail, data sources that drift, dependency and runtime versions that differ across environments, permissions and rate limits, floating-point behavior, and infrastructure failures. Any of these can make a nominally deterministic script behave inconsistently or fail in production. The honest claim is therefore narrower than “each step becomes 100% reliable”: moving work into the Execution layer restores reproducibility under controlled conditions

and removes the step as a source of stochastic model error, but operational reliability is an additional engineering achievement.

That achievement comes from disciplines layered on top of determinism: input validation, explicit error handling, idempotent operations, dry-run-before-commit safeguards, and governance over external dependencies. The case study in Section 7.6 illustrates these disciplines in a deployed skill, and Section 8 treats them as learning objectives in their own right. Reliability, in short, is determinism plus validation, error handling, and governance—and teaching students to tell those parts apart is itself a core architectural competency.

The Skill Pattern: Portable, Self-Contained AI Capabilities

Anatomy of a Skill

A skill is a self-contained folder that encapsulates everything an AI agent needs to perform a task: a directive (SKILL.md) describing what the skill does and when to invoke it, a scripts/ subdirectory of deterministic Python implementations, and a references/ subdirectory containing schemas, API docs, and examples the orchestrator may need.

This structure imposes three properties:

Portability. A skill folder can be moved between projects, shared across teams, or published as a reusable component. The skill carries its own context — no external dependencies on configuration buried in someone else’s code.

Encapsulation. The directive specifies exactly what the skill does, the scripts are independently testable, and the references are local to the skill’s scope of work.

Composability. Skills can invoke other skills, enabling complex workflows to be built from simple, well-tested components.

Building Skills: The Process

Building a skill follows a consistent six-step pattern:

1. **Identify a repeating workflow.** What operation do you perform repeatedly? What task consumes time that could be automated?
2. **Decompose into layers.** For each step, determine whether it requires probabilistic reasoning (Orchestration) or deterministic computation (Execution).
3. **Write the directive first.** Before writing code, describe the skill’s purpose, triggers, procedure, and expected outputs in SKILL.md.
4. **Implement the scripts.** Write the Python scripts for the Execution operations identified in step 2. Each script should be independently testable.
5. **Assemble references.** Gather schemas, examples, and API documentation that the orchestrator may need to invoke the skill correctly.
6. **Test and refine.** Run the skill on representative inputs, observe failures, apply the self-annealing loop, and iterate.

Composability and Workflow Construction

Individual skills are building blocks. Architectural power emerges when skills are composed. A financial-reporting workflow might chain four skills:

1. A data-extraction skill that pulls from multiple sources using deterministic database connectors and API clients.
2. A financial-analysis skill that computes ratios, trends, and variances arithmetically.
3. An anomaly-detection skill where the orchestrator reasons (probabilistically) about which metrics warrant attention.
4. A report-generation skill that produces formatted output deterministically.

The orchestrator passes outputs between skills, handles errors, and presents results to the user. As libraries grow, skills can be organized by domain, complexity, or system layer—no single scheme is universally correct, and the choice should reflect how the library is used and discovered.

AWS Bedrock as Enterprise Orchestrator

AWS Bedrock provides a managed foundation-model service that maps directly to the three-layer architecture and makes the separation visible: students see Directive, Orchestration, and Execution as distinct configurable components. Bedrock Agents are configured with **instructions** (Directive), **action groups** (Execution — Lambda functions or API calls), and a foundation model (Orchestration); the agent invokes action groups based on the instructions and the user’s request—exactly the pattern of probabilistic reasoning calling deterministic code. Its Knowledge Bases reinforce a key production discipline: authoritative facts live in retrievable documents, not in model weights, with the LLM interpreting queries and synthesizing responses against verified sources. Alternative platforms (LangChain, OpenAI Assistants API, CrewAI, AutoGen) can serve the orchestration role; Bedrock is recommended for teaching because its architectural boundaries are explicit in the configuration UI.

Python as the Deterministic Execution Layer

Python is recommended for the Execution layer based on its readability, library ecosystem, and widespread adoption in business and data analysis. Students who learn Python in this context develop a skill that transfers across IS roles.

Python’s libraries cover every domain relevant to IS education: boto3 for AWS, pandas for tabular data, requests for APIs, sqlalchemy for databases, openpyxl for spreadsheets, matplotlib for visualization.

The critical property is reproducibility: for the operations that matter in business contexts, Python produces the same output for the same input under controlled conditions. This makes scripts testable, auditable, and trustworthy in ways LLM outputs cannot be—though, as Section 3.5 cautions, reproducibility supports operational reliability without guaranteeing it.

Domain Applications

The three-layer skills architecture applies across IS curriculum domains. §7.1 walks through a representative skill in detail; §§7.2–7.4 sketch parallel skills in other domains; §7.6 grounds the framework in a skill running in production.

Database Management: The Query Validator Skill

Students build a skill that evaluates SQL queries for correctness, performance, and business alignment. **Execution scripts** handle syntax parsing (sqlparse), execution plan analysis (EXPLAIN ANALYZE via sqlalchemy), and index coverage checking against the information schema—all deterministic. The **orchestrator** interprets the execution plan in business context: a full table scan on a 10-row lookup table is acceptable; on a 10-million-row transaction table, it is not. Students learn that the “easy” part (running EXPLAIN) belongs in deterministic code while the “hard” part (contextual performance judgment) belongs in the probabilistic layer—mirroring how database administrators actually work.

Accounting and Financial Analysis: The Financial Auditor Skill

Students build a skill that retrieves financial data, computes standard ratios, and flags anomalies against industry benchmarks. Probabilistic interpretation of which anomalies to investigate; deterministic Python for the ratio calculations and benchmark comparisons.

Business Intelligence: The Data Pipeline Skill

Students build a skill that extracts data from multiple sources, applies transformations, and produces visualizations. Probabilistic decisions about which views best answer the stakeholder's question; deterministic Python for ETL and chart generation.

Project Management: The Sprint Analyzer Skill

Students build a skill that retrieves project data from Jira, computes velocity metrics, and identifies at-risk stories. Probabilistic reasoning about risk patterns and stakeholder language; deterministic Python for velocity calculations and Jira API calls.

The Cross-Domain Pattern

Across all four domains, the same architectural pattern holds: the directive frames the task in natural language, the orchestrator decides which scripts to invoke and how to interpret results, and Python scripts handle the operations where determinism matters. Table 1 maps this pattern across the four domains.

Table 1. The cross-domain three-layer pattern.

Component	Database	Accounting	BI	Project Management
Directive	Query review SOP	Financial analysis SOP	Data pipeline SOP	Sprint review SOP
Orchestration	Interpret execution plans in business context	Interpret ratios in industry context	Determine what analysis matters	Interpret risks in team context
Execution	EXPLAIN, index checks, syntax validation	Ratio computation, benchmark comparison	ETL, statistics, visualization	API data retrieval, velocity computation
Deterministic operations	Parse, query, compute	Calculate, compare, flag	Extract, transform, chart	Fetch, compute, threshold
Probabilistic operations	Contextual performance judgment	Business interpretation	Analysis selection, narrative	Risk contextualization

This consistency is the framework's greatest pedagogical strength. Students who learn the pattern in one domain can apply it in any other.

A Deployed Example: The research-project-sync Skill

The domain skills above are illustrative. To show that the architecture is not merely a teaching abstraction, we present a skill that runs in production. The academic portfolio site of one of the authors uses a research-project-sync skill to keep the research-project cards shown to visitors in agreement with per-project source files. It is deployed on a live server, invoked automatically by the deployment pipeline, and built on exactly the three layers this paper describes. Table 2 maps the deployed skill onto the framework.

Table 2. The research-project-sync skill mapped to the three layers.

Layer	Artifact in research-project-sync	Role
Directive	SKILL.md	Declares the trigger (“after edits to any project source file”), the source-of-truth rule, the script inventory, and explicit safety constraints. Read by both humans and the orchestrator; never executed.
Orchestration	The LLM (or, in production, the deployment trigger)	Reads the directive, decides which script to run with which flags, interprets results, and requires confirmation before any destructive change.
Execution	list_projects.py, preview_sync.py, sync_projects.py, validate.py	Deterministic Python: read current state, diff source against the database, upsert by key, and assert post-conditions. No reasoning—only computation.

The skill embodies the Section 3.5 distinction between determinism and operational reliability through concrete safeguards rather than assumptions. `sync_projects.py` is dry-run by default: without an explicit `--confirm` flag it prints the planned changes and exits without writing. Deletion is opt-in behind a separate `--delete-stale` flag, so an accidentally removed source file cannot silently erase live content. The shared database layer fails loudly on missing credentials—exiting with a distinct error code rather than substituting defaults—and `validate.py` asserts cross-checks (every source file has a database row and vice versa; every URL is well-formed) and returns a non-zero exit code on any violation. The upsert is idempotent: running it twice yields the same end state. These are precisely the validation, error-handling, and governance disciplines that turn a deterministic script into a reliable one. As a teaching artifact the skill is immediately usable: a real directive, a real script inventory, and real safety constraints that students can read, run, modify, and extend—matching the progressive-complexity model of Section 8.3.

Pedagogical Implications

Teaching Architectural Thinking

The three-layer skills framework teaches a mode of thinking that differs from both traditional programming (write deterministic code) and current AI usage (write prompts). It asks students to simultaneously hold the strengths and limitations of two paradigms and design the interface between them.

Decomposition — determining what belongs in which layer — is the central learning objective. It requires students to reason about reliability, predictability, and the cost of error for each step.

Assessment Through Skill Building

The skills framework provides a natural assessment model that addresses the integrity concerns of evaluating student work in the AI era. Students submit complete skills — directive, scripts, references — and are graded on four dimensions:

- **Decomposition:** did the student correctly identify which operations belong in Orchestration versus Execution?
- **Directive quality:** is the directive clear, specific, and complete on triggers, procedure, and edge cases?
- **Script correctness:** do the Python scripts produce correct, deterministic results? Scripts are independently testable.
- **System integration:** does the complete skill work end-to-end?

These four dimensions can be operationalized as an analytic rubric (Table 3). Each is scored on three levels—*Beginning*, *Developing*, and *Proficient*—giving students explicit targets and graders a consistent instrument.

Table 3. Analytic rubric for skill-building assessment.

Dimension	Beginning	Developing	Proficient
Decomposition	Operations assigned to the wrong layer; probabilistic and deterministic work conflated	Most operations correctly placed; boundary cases misassigned	Every operation justified by its reliability requirement; ambiguous cases reasoned about explicitly
Directive quality	Vague purpose; missing triggers or procedure	Clear purpose and procedure; edge cases or safety constraints omitted	Precise triggers, procedure, references, and explicit safety constraints
Script correctness	Scripts fail or return wrong results; not testable	Scripts work on typical inputs; weak validation and error handling	Correct, deterministic, independently tested, with input validation and explicit error handling
System integration	Components do not work together end-to-end	End-to-end path works for the happy path only	End-to-end skill works, including failure paths and the self-annealing loop

Because the graded artifact is a working system rather than a prose answer, the rubric is resistant to the integrity problems of AI-era assessment: a student cannot defend a decomposition they do not understand, and the scripts either run and pass tests or they do not.

Progressive Complexity

The framework supports a natural progression of complexity that maps to course levels:

Introductory: students use existing instructor-built skills and observe the three-layer architecture in action.

Intermediate: students modify existing skills by writing new scripts and extending directives.

Advanced: students architect new skills from scratch — identifying a workflow, decomposing it, writing the directive, implementing scripts, and assembling references.

This mirrors the trajectory from using technology to designing it.

Connection to Workforce Demands

The 56% wage premium for AI-skilled workers documented by PwC (2025) does not reward typing prompts into ChatGPT — it rewards the architectural skills required to deliver reliable, maintainable, governable AI systems at production scale: reliability through deterministic execution; maintainability through separation of concerns; governability through explicit documentation of behavior; and business alignment through domain-specific skill design.

Limitations and Future Research

This paper presents a conceptual framework and proposes pedagogical applications based on practitioner methodology rather than completed empirical research. Several limitations warrant explicit acknowledgment.

Empirical validation. The three-layer skills architecture and its pedagogical applications have not yet been tested through controlled implementation. Claims about learning outcomes, assessment reliability, and graduate readiness remain to be validated; we outline a concrete design for doing so below.

Platform specificity and practitioner origins. While the architectural principles generalize, the implementation guidance leans on AWS Bedrock and the practitioner-developed skill pattern. This skill

pattern (Saraev, 2025) — though robust in industry use — has not been subjected to formal pedagogical theory, and the framing offered here is one possible translation among several.

Evolving capabilities. The boundaries between what LLMs handle reliably and what requires deterministic code are moving. The framework itself is durable — the separation principle holds even as the line shifts — but specific recommendations may require revision as tools evolve.

A planned evaluation design. To move beyond conceptual argument, we outline a design-based evaluation that future work can execute. The design is quasi-experimental: two sections of an IS course—one taught with the conventional “use the AI tool” approach, one with the three-layer skills-building approach—are compared on common outcome measures. Because students cannot be randomly assigned to sections in practice, the design matches sections on prior GPA and a baseline pre-test and treats the comparison as quasi-experimental rather than a true experiment.

Three outcome measures operationalize the paper’s central claims: (a) a decomposition task scoring whether students correctly route operations to the probabilistic or deterministic layer; (b) a transfer task asking students to architect a skill in a domain not covered in class, scored with the Section 8.2 rubric; and (c) a diagnostic task measuring whether students can distinguish a deterministically wrong script from a correct one—operationalizing the Section 3.5 distinction between determinism and correctness. Analysis compares treatment and control on each measure, with inter-rater reliability established for rubric-scored artifacts.

Threats to validity are acknowledged explicitly: instructor effects, self-selection across sections, and the novelty effect of a new pedagogy. Replication across multiple terms and instructors, and pre-registration of the outcome measures, would partially address them. This design would yield the empirical evidence the present conceptual contribution does not yet provide, and it is the authors’ intended next step.

Conclusion

The central argument of this paper is that the professional competency of the AI era is not the ability to use AI tools but the ability to architect AI systems — to know what should be probabilistic and what must be deterministic, and to design the interfaces between them. The three-layer skills architecture provides a concrete, teachable framework: the Directive layer specifies intent, the Orchestration layer reasons probabilistically, and the Execution layer performs deterministic operations—packaged as portable, composable skills that students can build, assess, and reuse.

IS programs face a choice: teach students to use AI tools — a competency with a short half-life — or teach them to architect AI systems. The programs that choose the latter will produce the professionals who design the AI systems that organizations depend on.

Generative AI Acknowledgment

This paper was developed with the assistance of generative AI tools (Claude, Anthropic). AI was used as a writing and editing aid for drafting, restructuring, and refining prose. All intellectual contributions — including the three-layer skills architecture, domain applications, and all arguments presented — are the original work of the authors. The authors reviewed, validated, and take full responsibility for the originality, validity, and integrity of all content.

References

- Amazon Web Services. (2024). *Amazon Bedrock User Guide*. <https://docs.aws.amazon.com/bedrock>
- Biggs, J. (1996). Enhancing teaching through constructive alignment. *Higher Education*, 32(3), 347-364. <https://doi.org/10.1007/BF00138871>
- Brown, T. B., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901.
- Chase, H. (2023). LangChain: Building Applications with LLMs through Composability. [Computer software]. <https://github.com/langchain-ai/langchain>
- Fan, A., Lewis, M., & Dauphin, Y. (2018). Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 889-898. Association for Computational Linguistics. <https://doi.org/10.18653/v1/P18-1082>
- Firth, D. R., Derendinger, M., & Triche, J. (2024). Cheating better with ChatGPT: A framework for teaching students when to use ChatGPT and other generative AI bots. *Information Systems Education Journal*, 22(3), 47-60.
- Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The curious case of neural text degeneration. In *Proceedings of the International Conference on Learning Representations*. <https://openreview.net/forum?id=rygGQyrFvH>
- Kolb, D. A. (1984). *Experiential learning: Experience as the source of learning and development*. Prentice Hall.
- Marshall, B. (in press). AI-augmented IS curriculum: A comparative analysis of model curriculum evolution and pedagogical transformation. In *Proceedings of the International Association for Computer Information Systems Conference*.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058. <https://doi.org/10.1145/361598.361623>
- PwC. (2025). The fearless future: 2025 global AI jobs barometer. <https://www.pwc.com/gx/en/services/ai/ai-jobs-barometer.html>
- Saraev, N. (2025, December 21). Agentic workflows course: The 6-hour masterclass [Video]. YouTube. <https://www.youtube.com/watch?v=MxyRjL7NG18>
- Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.
- Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998-6008.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., et al. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 186345.

<https://doi.org/10.1007/s11704-024-40231-1>

Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. Microsoft Research Technical Report.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations (ICLR).