DeepDomain AI: System Design and Validation for Enterprise AI Talent Training

Pengyue He, Suya Cheng*, Weili Wang, Lulu Ma, Yutong Li
School of Artificial Intelligence and Big Data, Henan University of Technology, Zhengzhou, Henan, China
**Corresponding Author*

Abstract: Enterprise AI roles evolve faster than conventional course catalogs, while direct use of large language models (LLMs) for training-content generation introduces risks such as unsupported claims, non-executable code, and incomplete audit trails. This paper presents DeepDomain AI, a prototype platform for personalized enterprise AI training. Built with React 18 and FastAPI, the platform compares learner evidence with a structured catalog of 15 AI roles, identifies readiness gaps, and organizes role-oriented training paths. Its resource workflow produces foundation-oriented and project-oriented candidates, records structured review results, and applies a publish/correct/reject quality gate. Pydantic schemas constrain model outputs, and a Docker sandbox scores code through task-specific executable assertions rather than source-code keywords. Functional tests cover job analysis, path generation, knowledge recommendation, state persistence, and sandbox execution. In an offline evaluation of 45 constructed learner profiles, dynamic ranking achieved a Top-3 hit rate of 93.33%; on the 30 same-direction profiles used for comparison, the static-label baseline achieved 80.00%. A 20-sample threshold-consistency test produced a 100% interception rate for rule-violating resources and a 0% false-positive rate for rule-compliant resources. In this study, the system was tested with constructed learner profiles rather than data collected from employees. The results therefore describe how the implemented rules and ranking method performed in our test setting; they do not yet show whether the generated resources improve learning or meet expert expectations in practice.

Keywords: AI Workforce Training; Multi-Agent Collaboration; Learner Profile; Job Knowledge Graph; Resource Quality Gate; Hallucination Control

1. Introduction

AI-related job requirements now extend beyond model training to prompt engineering, retrieval-augmented generation (RAG), multi-agent orchestration, deployment, and security evaluation. LLMs can generate instructional material on demand, but enterprise training requires stronger controls than open-ended generation. Unsupported technical claims may mislead learners [1], and excessive adaptation to stated preferences may weaken the depth required by the target role. The practical problem is therefore not only how to personalize content, but how to connect personalization with verifiable role requirements and executable assessment.

Many conventional training platforms still use fixed videos and standardized quizzes. Learners also tend to follow the same sequence of activities. This model is relatively easy to operate, but it responds slowly when the requirements of a target role change. It also provides limited support when a learner needs remediation in a specific area.

A single LLM can answer questions or prepare draft materials, but these capabilities do not ensure reliable training content. Its responses may vary in structure, and some of the generated claims may be poorly grounded. Previous studies have investigated multi-agent frameworks [2,3]. Even so, many existing systems do not provide an integrated quality-assurance mechanism. Generated content may proceed without an explicit quality gate [4]. Learner-submitted code is also not always validated in an isolated execution environment.

DeepDomain AI was developed to examine this engineering gap. Its workflow begins with learner intake. The system analyzes the requirements of a target role and calculates the

gap between those requirements and the learner's current profile. It then builds a training path and initiates dual-track resource generation. A structured review determines whether the resulting materials can be used or require correction. Learners subsequently complete sandbox-based practice, while the state-archival component retains the intermediate results and later updates.

The role catalog defines the capabilities expected for each role. Evidence stored in the learner profile provides the basis for explaining readiness and setting training priorities [5-10]. Model-assisted work is divided across several roles. One role performs diagnosis, and another generates learning resources. Review and planning are handled separately. By contrast, deterministic scoring and schema validation remain outside free-form model control. This separation is important because the model output is treated as an intermediate artifact that must be checked, rather than as a final system decision.

Programming tasks follow a different validation process. DeepDomain AI runs learner-submitted code in an isolated Docker environment. Executable assertions then inspect its actual behavior instead of relying only on the submitted text or a model-generated explanation [11,12].

This paper presents the design and implementation of these mechanisms. Offline experiments are used to examine the functional behavior of the prototype and the performance of its role-ranking method. Comparative tests further reveal several limitations of the current implementation. These results remain limited to the tested profiles and experimental settings, so the system's performance with a broader learner population requires further validation.

2. Requirements Analysis

2.1 Overall Requirements

DeepDomain AI supports employees moving from conventional technical positions into AI-related roles. The system converts changing role requirements into structured, machine-processable skill indicators and proficiency levels. It compares these requirements with evidence from the learner profile and identifies capability gaps. The resulting gaps determine the order of training units and their observable deliverables. The system also retains the provenance of model-assisted decisions for later inspection.

The workflow begins with learner intake. Job analysis and diagnosis identify the requirements of the selected role and the learner's current readiness. Path generation then organizes the required training units. Resource learning and practical assessment provide additional evidence, which is retained during archival.

Learners submit a target direction, self-reported skills, and previous project experience. After selecting a role, they receive a gap-oriented plan. They complete quizzes and project challenges before moving to sandbox exercises and mock interviews.

Training specialists maintain the knowledge resources and configure agent workflows. They use the specialist interface to inspect cohort progress and review compliance records. The available records include de-identification checks and model interactions. Resource-review decisions and correction logs are also retained. Figure 1 shows the resulting business loop.

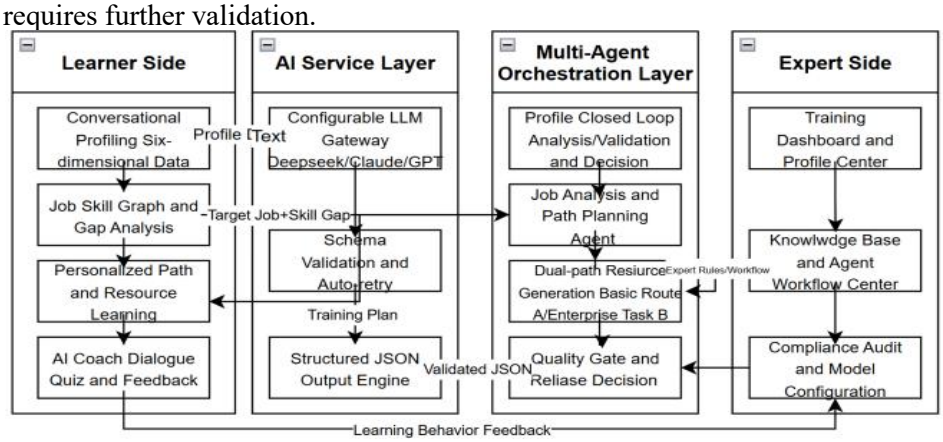

Figure 1. Business Closed Loop of DeepDomain AI for Enterprise AI Skill Training

2.2 User Roles

The platform serves learners and enterprise training specialists (Figure 2).

Learners enter skills and experience, inspect role requirements, select a target role, and follow a

personalized training plan. Their profile records progress, capability changes, project artifacts, interview results, and completion evidence.

Training specialists monitor cohorts, compare learner groups, maintain knowledge documents and chunks, configure agents and model accounts, and inspect de-identification, interaction, resource-review, and correction logs. The functions available to each type of user are listed according to their responsibilities (Table 1).

Table 1. Mapping Between User Roles and Functional Requirements

Role	Primary needs	Modules
Learner	Select a target role, address capability gaps, and follow an executable training path	Learner home, job graph, training path, knowledge resources, sandbox
Training specialist	Monitor cohort capability, review resources, and configure agent workflows	Specialist cockpit, profile management, knowledge base, agent management, compliance

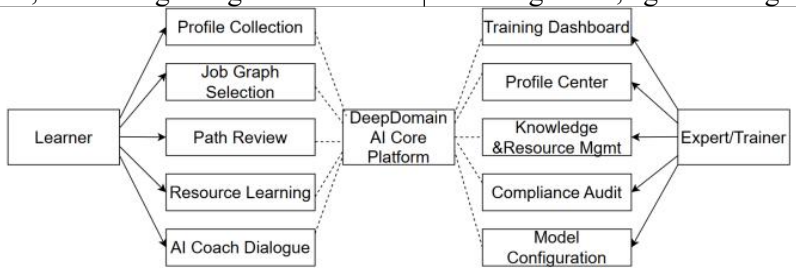

Figure 2. User Roles and Functional Relationships

2.3 Functional Requirements

Job analysis uses the learner's selected skills, target direction, and preferred work content. Practice level and project experience help interpret this information. Answers to the objective and open-ended questions are included as evidence. Based on these inputs, the analysis produces a set of candidate roles. Each result is accompanied by supporting evidence, together with an account of the learner's strengths, risks, capability gaps, and training priorities.

After the learner selects a role, the system creates an ordered set of training units. Each unit has a defined objective, difficulty level, and duration. Depending on the training need, the assigned resource may take the form of a lecture, guide, quiz, mind map, video script, or training agent.

Practice begins with staged question banks. Integrated challenges are introduced later, followed by real-task evaluation and sandbox execution. Mock interviews provide another form of assessment. The score from each activity is written to the learner record with an explanation. Identified strengths are retained in the same record, while improvement suggestions influence later recommendations.

The specialist interface presents cohort-level statistics, with filters for locating individual learners. A specialist can open two profiles side by side when a comparison is required. Knowledge resources are maintained through a separate management view. Compliance reviews can also be opened from the interface. The associated correction history remains available for inspection. Knowledge documents are divided into chunks, which serve as the retrieval basis for generated resources.

The multi-agent workflow assigns diagnosis and path planning to separate stages. Resource generation considers two planning perspectives. Another stage reviews the resulting candidates and requests corrections when necessary. Each stage retains the original input and generated output. Its record also shows the current status and execution time. When a correction occurs, the relevant details are added to that record. This information supports later inspection of the workflow instead of leaving only the final response.

2.4 Non-Functional Requirements

Reliability requirements address model timeouts, invalid responses, generation failures, and schema-validation errors. Long-running plan and resource jobs move from pending to running before ending as completed or failed. Invalid output cannot advance the workflow.

Pydantic schemas constrain field types, lengths, enumerations, and numeric ranges. Model-produced JSON must pass validation. Failure triggers bounded retries with field-level feedback; repeated failure terminates the task explicitly.

Model credentials are encrypted at rest and

masked in the interface. Untrusted text is separated from system instructions, while compliance records support subsequent permission and data-governance controls.

For traceability, the system establishes a complete audit trail for each model invocation and multi-agent workflow, recording the provider, model name, invocation duration, workflow node, input summary, output content, and correction result. These records are persisted so that specialists can trace the origin and processing history of generated results and use them for problem review and quality analysis.

Maintainability is supported by front-end/back-end separation, modular pages and services, typed interfaces, and structured cross-module contracts. These data objects are managed by the corresponding system services (Table 2).

Table 2. Principal Data Objects and Service Boundaries

Data object	Key fields	Interfaces/services
Learner profile	id, name, background, target, match_score, strengths, weak_points	profile, diagnose, learner_home
Job capability graph	id, title, skills, edges (source, target)	job_ graph, job_analysis
Training resource	title, content, target_role, knowledge_points, match_rate, version	resources, resource_quality_pipeline
Agent pipeline	agent_id, role, node_label, status, input_text, output_text, duration_ms	agents, agent_orchestrator, debate_service
Sandbox task	title, initial_code, solution_code, assertions (input, expected)	Sandbox_service, sandbox API

3. System Architecture

3.1 Deployment Architecture

The deployment design contains browser clients, a reverse proxy, front-end and back-end applications, a model gateway, model services, a data node, and an isolated sandbox node (Table 3 and Figure 3). Containerized components reduce configuration drift between development and deployment environments.

Table 3. Deployment Nodes and Technology Stack

Node	Deployment	Technology	Network exposure
Browser	Learner workstation	Chrome, Edge, or equivalent	Internet egress
Reverse proxy	Container	Nginx	Internet ingress
Front end	Container	React and Vite static build	Internal network only
Back end	Container	FastAPI asynchronous web service	Internal network only
Model gateway	Container	Python routing and rate limiting	Internal network; proxied egress
LLM service	Self-hosted or managed	vLLM or commercial LLM service	Internet egress
Data node	Container	SQLite	Internal network only
Sandbox node	Container	Isolated Python runtime	Isolated network segment

External traffic enters through the application network, whereas code execution is confined to an isolated sandbox network. The sandbox has no direct database or external-network access and uses input isolation, a read-only filesystem, capability removal, process and memory limits, and execution timeouts. Runtime dependencies are placed in a versioned image rather than installed by learner submissions.

The prototype runs on a single workstation. A production deployment would separate front-end and back-end services, migrate the database to managed storage, and expose environment-specific endpoints through configuration rather than code changes.

3.2 Software Architecture

The software follows a separated front-end/back-end design with explicit responsibility boundaries (Figure 4).

The React front end follows the structure shown in Figure 4. Pages use reusable components, while API clients and state hooks connect the interface to application data. Styles and utilities are shared across the views. TypeScript checks the interfaces statically. Vite produces builds for development and production. Interaction components come from Ant Design, and ECharts draws the job graphs, capability radars, progress trends, and cohort statistics.

The FastAPI back end keeps routes separate from business services. Schemas define the data exchanged across these boundaries; persistence models represent the stored records. Infrastructure and seed data remain in separate modules. Pydantic models validate API requests

and responses before they reach the business logic. SQLAlchemy writes and retrieves records through the persistence layer. When a route raises an exception, the centralized handler converts it into a consistent error envelope. A test can target one business service through its schema-defined boundary. Unrelated modules remain unchanged.

Long-running model-assisted operations return a task identifier for client-side progress polling. Diagnosis outputs, candidate resources, review results, and correction records remain available to specialists. Retaining these stage-level artifacts makes asynchronous model operations auditable.

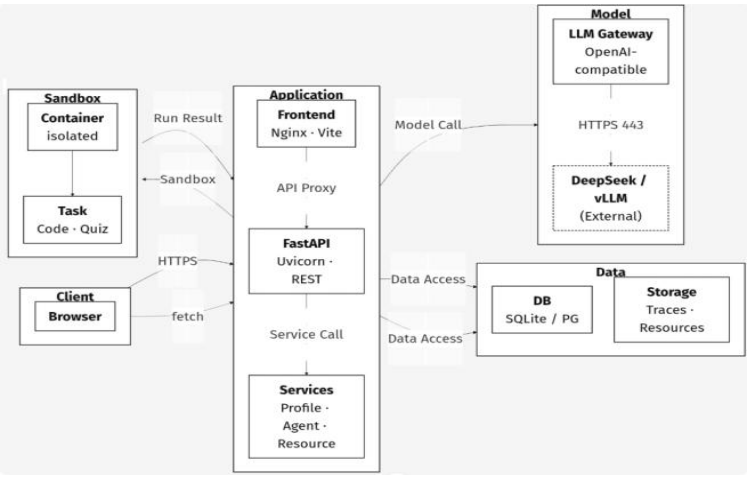

Figure 3. Containerized Deployment Architecture

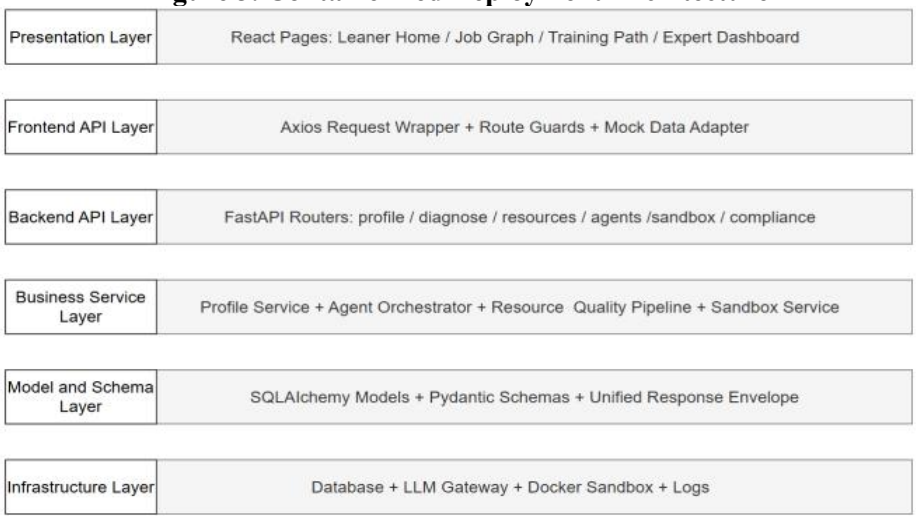

Figure 4. Front-End/Back-End Layers and Service Boundaries

3.3 Core Modules

Data enters the core modules through learner intake and then passes to diagnosis. The diagnostic output is used for planning before resources are generated and reviewed. Practice results are archived at the end of the workflow (Figure 5). Typed schemas specify the inputs and outputs at each step.

The learner-profile module stores background, target role, skills, preferences, and assessment evidence. Job analysis and path planning read these structured fields.

The job-graph and path-planning module represents role skills and their dependencies. Readiness gaps are calculated against the learner profile. The module uses those gaps to place foundational units before project practice and integrated assessment.

The mapping works in both directions. Role requirements expose gaps in the current profile; evidence from later training updates it. In the current implementation, direction and preferred work content carry more weight in candidate ordering. Skill readiness mainly determines the learner's starting point in the training path.

The agent module manages role definitions, model bindings, workflows, execution states, and debate or review records. Asynchronous calls may run independent generation branches concurrently, and each stage is persisted for

inspection.

The quality-gate module evaluates generated resources through content-quality and personalization indicators, then applies a publish, correct, or reject decision according to configured thresholds (Section 4.4).

Structured review records retain metrics, comments, corrections, and provenance. This provides an auditable basis for expert inspection, although the present study does not establish that multi-agent review outperforms single-agent review on independently labeled content.

The sandbox and compliance module executes learner code in isolation and records checks related to personal information, API security, resource content, and model outputs.

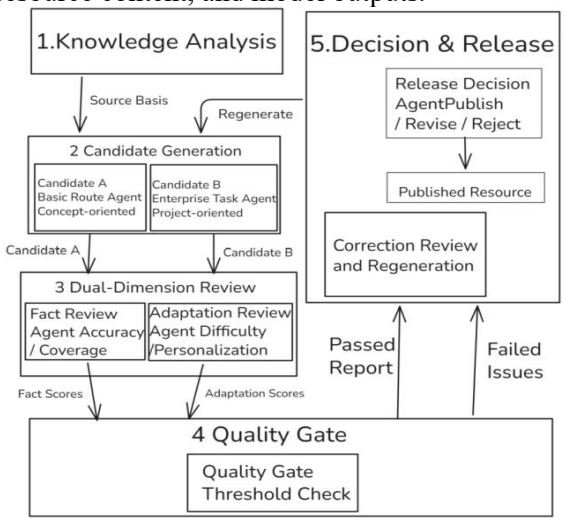

Figure 5. Multi-Agent Resource-Generation and Quality-Gate Workflow

4. Implementation and Key Techniques

4.1 Front-End Interaction and Visualization

The learner interface follows the sequence of selecting a role, inspecting the job graph, generating a path, studying resources, completing sandbox tasks, and reviewing outcomes. The specialist interface emphasizes cohort overview, profiles, knowledge management, agent configuration, and compliance. Visualizations are used where relationships or change over time are central to the task.

4.2 Back-End Services and Unified Contracts

FastAPI supports the I/O-bound model calls used by the orchestration layer. Routes cover learners, jobs, training, resources, profiles, knowledge, compliance, model configuration, and sandbox execution. Pydantic models define learner profiles, resource reviews, and stage results, reducing ambiguity across modules and allowing tests to assert business fields directly.

4.3 Job Graph–Profile–Training Path Mapping

Each role defines its knowledge points and skill requirements. Practical tasks and evaluation targets are recorded in the same role profile. The system compares these fields with the learner profile to identify and prioritize gaps (Figure 6). Path planning respects prerequisites and time constraints. Learner preferences and role importance then affect the order, which resembles constrained ordering on a directed acyclic graph. In a RAG path, document processing leads to retrieval and reranking, followed by evaluation and monitoring.

The mechanism does not depend only on manually assigned learner tags. Role direction and preferred work content contribute to the ranking, along with readiness evidence. However, the ablation test produced the same aggregate hit rates after skill coverage was reduced from complete coverage to approximately 60%, because skill readiness is not used as a direct key in the final ranking order.

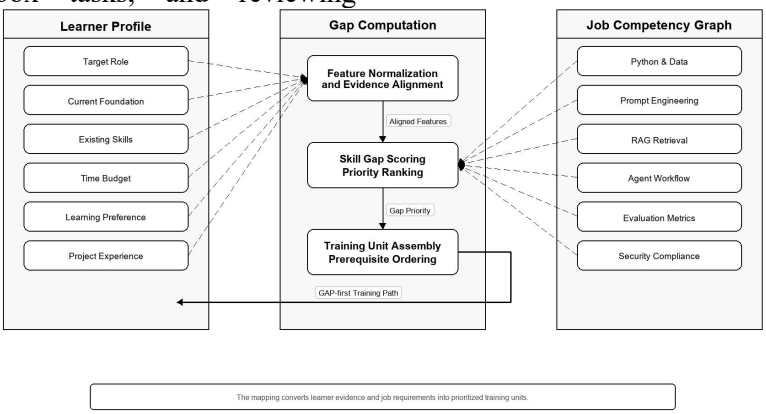

Figure 6. Mapping between the Role Capability Graph and Learner Profile

4.4 Multi-Agent Orchestration

The orchestrator resolves the model configuration used for diagnosis, resource generation, review, and path planning. For each stage, it records the state and execution time. The input summary and output are stored in the same record. These records support the progress display and later reconstruction of the execution path.

Resource generation produces two candidates. One follows a foundation-first plan; the other starts from project work. The review measures factual accuracy and knowledge coverage, then checks the hallucination rate and difficulty match. Personalization and goal alignment complete the six quality indicators. Based on these results, the gate returns publication, correction, or rejection. A corrected resource enters another review cycle.

Intermediate decisions remain visible instead of being reduced to an opaque final model response. The quantitative evaluation in this paper covers only the consistency of deterministic thresholds. Whether multi-agent review improves factual quality remains future work. Every generated resource is checked against the six indicators before publication (Table 4).

Table 4. Resource Quality-Gate Indicators

Indicator	Definition	Threshold
Factual accuracy	Consistency with the knowledge base and role standard	>= 90
Knowledge coverage	Coverage of unit objectives and key concepts	>= 90
Hallucination rate	Share of unsupported, exaggerated, or incorrect claims	<= 5%
Difficulty match	Fit with learner foundation and available time	>= 85
Personalization	Use of learner goals, preferences, and gaps	>= 85
Goal alignment	Contribution to target-role capability	>= 85

The thresholds used here are engineering defaults. A production study should calibrate them on a versioned validation set with blinded human ratings, rather than rely on model self-evaluation.

4.5 Schema-Constrained Output and Bounded Retry

After converting the target Pydantic models into JSON Schema instructions, the system checks returned content for JSON syntax, field types, enumerations, and required values. Any validation errors are passed back in a bounded retry request; if the response continues to fail validation, the task ends with an explicit error. This reduces ad hoc parsing across services, but schema compliance cannot establish factual correctness.

The schema layer converts otherwise free-form model responses into typed objects. The front end and back end share these objects, while the audit log and regression tests use the same structure.

4.6 Sandbox Training and Compliance

Sandbox tasks comprise a template, initial code, learner submission, runtime result, and feedback (Figure 7). Untrusted code runs in an isolated container with restricted resources. Compliance checks cover sensitive fields, abnormal responses, permission boundaries, and log retention.

Network isolation, runtime restrictions, and output auditing address different risk categories. The current validation confirms selected controls and service behaviors; it does not include a complete container-escape or high-concurrency security assessment.

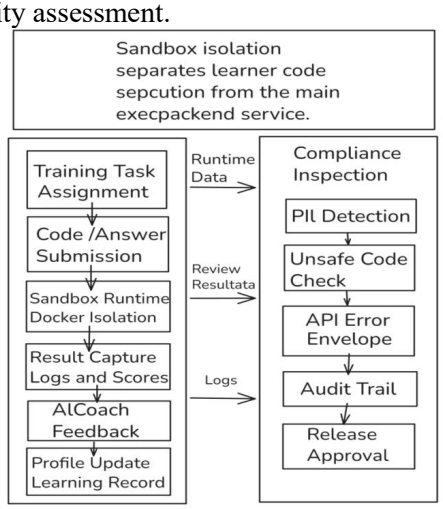

Figure 7. Sandbox Training and Compliance Workflow

5 Evaluation

5.1 Environment

The prototype uses React 18.3.1, TypeScript 5.5.4, and Vite 5.4.7 on the client, and FastAPI, Pydantic, SQLAlchemy, and Uvicorn on the server. SQLite supports prototype persistence.

Docker Desktop in Linux-container mode runs a python: 3. 12-alpine image with Flask, NumPy, and FAISS. Learner code executes without network access and under resource limits. The software versions and runtime settings used in the evaluation are given in Table 5.

Table 5. Evaluation Environment and Configuration

Layer	Configuration	Purpose
Front end	React 18.3.1, TypeScript 5.5.4, Vite 5.4.7	Rendering and API integration
Back end	FastAPI, Pydantic, SQLAlchemy, Uvicorn	Service orchestration and schema validation
Data	SQLite and isolated test databases	Persistence and state-transition checks
Sandbox	Docker Desktop; python:3.12-alpine	Execution against published assertions

5.2 Functional Validation

Five behaviors were tested: assertion-gated sandbox scoring; safe degradation after model failure; immunity of deterministic scoring to prompt injection in free-text experience; responsiveness of readiness evidence and, in selected cases, ranking to skill changes, and persistence of paths, mastery, progress, and credentials. Test inputs included 15 AI role profiles, two 12-question direction assessments, learner profiles, isolated SQLite databases, and 12 published sandbox tasks with executable assertions.

5.2.1 Assertion-gated sandbox scoring

Each published task is paired with executable Python assertions. The submission and task-specific assertions run in Docker with no network, a read-only filesystem, dropped capabilities, a 64-process limit, 256 MB of memory, and a 12-second timeout. Scores therefore depend on runtime behavior rather than the presence of expected names or keywords.

For an agent-tool task, assertions require callable entries and check concrete results. For a RAG-pipeline task, chunk_text (), keyword_search (), and rank_results () must exist and behave correctly. A submission containing only print ("looks fine") received a failed status and zero points. An unknown task identifier returned an unavailable status rather than an inferred score. Published reference solutions were separately checked against their assertion blocks.

5.2.2 Safe degradation under model failure

Injected providers simulated timeout, invalid JSON structure, and explanation-generation failure. In each case, deterministic scoring remained available, the analysis status became partial, and analysisSource recorded whether the result was rules_only or hybrid. The system did not fabricate missing explanations.

5.2.3 Prompt injection does not alter deterministic scores

Adversarial instructions were repeated in the free-text experience field and compared with a benign baseline. Candidate order, readiness score, and recommendation level remained identical because deterministic scoring consumes validated structured fields rather than concatenating the free text into its calculations.

5.2.4 Evidence-responsive job analysis

A profile with Python, FastAPI, RAG, and vector-database evidence favored RAG-related roles; replacing the RAG evidence with LangGraph and agent skills favored an agent-application role. A separate profile with SQL, Pandas, and visualization skills retained a data-analyst candidate even when its selected direction was broader AI application development. These cases show that evidence changes analysis outputs, while the aggregate ablation in Section 5.4 clarifies that moderate skill-coverage changes do not necessarily alter the final ranking.

5.2.5 Persistence across service boundaries

Generated learning paths retained their phases and deliverable names after retrieval. Knowledge recommendations included citation and version fields and respected prerequisites. Practice results updated mastery and stage eligibility. Model API keys remained encrypted at rest and were not exposed through list endpoints. The main results of the functional tests are presented in Table 6.

Table 6. Summary of Functional Validation Evidence

Validated behavior	Method	Evidence
Assertion-gated sandbox scoring	Execute learner code with published Python assertions in Docker	print("looks fine") scores 0 on task-rag-1; unknown task returns unavailable
Safe degradation after model failure	Inject timeout, invalid-schema, and explanation-failure providers	All return partial; analysisSource retains provenance
Prompt-injection resistance	Repeat adversarial instructions four times in experience text	Ranking and readiness remain identical to benign baseline

Evidence-responsive analysis	Compare profiles with different skill sets and identical assessment answers	RAG evidence favors RAG role; agent evidence favors agent role
Persistence consistency	Save and retrieve paths, mastery, progress, and model accounts	Stages and citations persist; gates execute; credentials remain encrypted

The functional tests establish implementation behavior. The following scenario then connects learner intake, diagnosis, planning, recommendation, practice, and state updates.

5.3 Application Scenario: RAG Application Engineer

Learner profiles are mapped to job requirements to identify capability gaps and rank candidate roles (Figure 8).

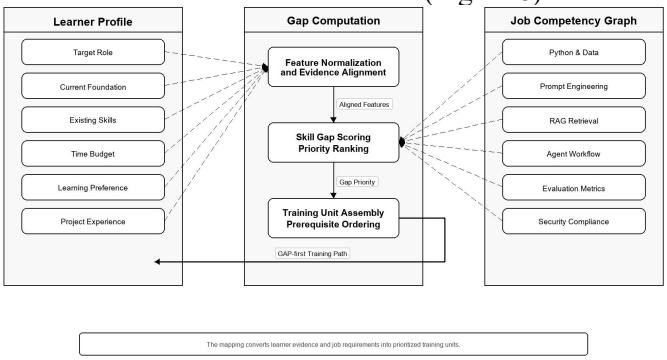

Figure 8. Mapping from Learner Profile to Job Requirements

The scenario profile records introductory experience with Python, FastAPI, RAG, and vector databases. The learner is interested in knowledge-base construction and AI application development. The ranking places rag-application-engineer first, followed by llm-application-engineer. When RAG-related skills are replaced with agent-development evidence, agent-application-engineer moves to the leading position, consistent with Section 5.2.4.

After role selection, the pipeline builds a phased path with named objectives and deliverables. The RAG path moves from retrieval-enhanced generation to recall evaluation and experiment logging. An evaluation notebook serves as its principal artifact. Recommendations carry citations and respect prerequisites, with greater weight assigned to diagnosed weak areas. Results from practice and the sandbox update mastery, which influences later recommendations and stage eligibility (Figure 9).

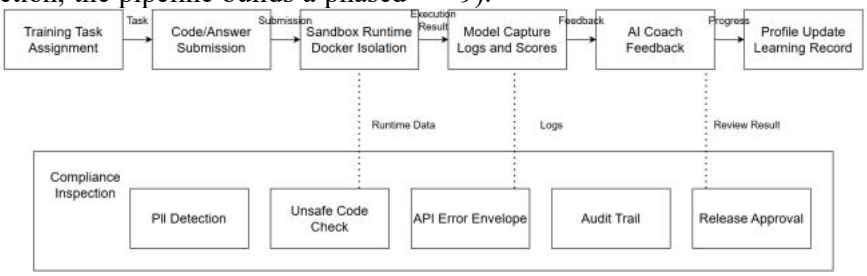

Figure 9. Sandbox Practice, Feedback, and State Updates

The prototype feedback loop begins with intake. Job diagnosis feeds path generation, followed by knowledge recommendation and practice. The resulting state is then written back (Figure 10). Each stage leaves an inspectable artifact rather than only a final dashboard value.

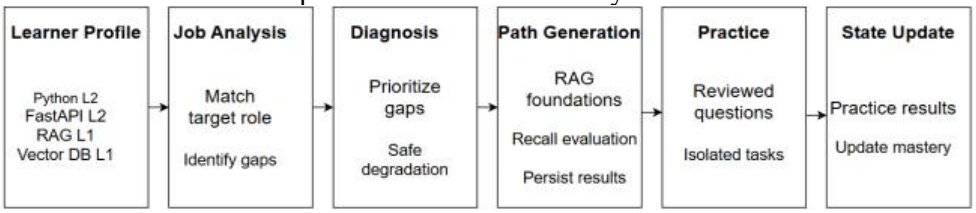

Figure 10. End-to-End Validation Scenario for RAG Application Engineer Training

5.4 Quantitative Offline Evaluation

Sections 5.2 and 5.3 address functional correctness and integration. This section uses constructed offline samples to evaluate recommendation ranking, verify quality-gate threshold consistency, and measure deterministic function latency.

5.4.1 Job recommendation hit rate

Forty-five learner profiles were derived from the 15-role catalog: complete skill match, approximately 60% skill coverage, and cross-direction profiles. The rule-based scorer ran with ai_available=False. A hit occurs when the expected role appears at rank 1 or within the first three results (Table 7).

Table 7. Job recommendation hit rates

Profile type	Samples	Top-1 hits	Top-3 hits	Top-1 rate	Top-3 rate
Complete match	15	8	14	53.33%	93.33%
Partial match	15	8	14	53.33%	93.33%

Complete and partial profiles produced identical aggregate hit rates because direction match and preferred work content dominate ordering, while skill readiness determines the training starting point. Cross-direction profiles scored zero because the selected direction filters the expected role from the candidate set.

A static-label baseline filtered roles by direction and retained their original catalog order. On the 30 same-direction profiles, its Top-1 and Top-3 hit rates were 26.67% and 80.00%, whereas dynamic ranking achieved 53.33% and 93.33% (Table 8). The gains were 26.67 and 13.33 percentage points. The unchanged results between complete and partial skill profiles indicate that the observed ranking gain arises primarily from work-content preference and dynamic ordering, not from the magnitude of the skill gap.

Table 8. Dynamic ranking versus the static-label baseline

Method	Valid samples	Top-1 rate	Top-3 rate
Static-label baseline	30	26.67%	80.00%
Dynamic bidirectional mapping	30	53.33%	93.33%

5.4.2 Resource quality-gate consistency

Twenty structured review samples were constructed from the six thresholds in Table 4. Ten satisfied every threshold; ten violated one or two thresholds. The deterministic gate assigned publish, correct, or reject, and the outputs were compared with the rule-derived labels (Table 9).

Table 9. Quality-Gate Interception and False-Positive rates

Type	Samples	Publish	Correct	Reject	Interception	False positive
Compliant resources	10	10	0	0	—	0.00%
Violating resources	10	0	4	6	100.00%	—

All ten violating samples were corrected or rejected, while all ten compliant samples were published. The resulting 100% interception and 0% false-positive rates demonstrate implementation consistency with the same rules used to construct the labels. They do not estimate performance on independently annotated real resources.

5.4.3 Deterministic latency

Across 45 job-scoring calls, mean latency was 0.738 ms and median latency was 0.715 ms. Across 20 gate decisions, mean latency was 0.002 ms. These measurements cover deterministic local functions only, not complete LLM-assisted path and resource generation. The measured latency of the two deterministic stages is reported in Table 10.

Table 10. Deterministic-Stage Latency

Stage	Mean latency	Measurements
Job scoring	0.738 ms	45
Gate decision	0.002 ms	20

In summary, dynamic ranking improved Top-3 hit rate over the static-label baseline on the constructed same-direction profiles, while exposing a limitation: skill-coverage reduction did not affect aggregate ranking. The gate exactly reproduced its threshold rules on synthetic labels, and local deterministic execution remained below 1 ms.

6 Discussion and Future Work

6.1 Discussion and Conclusions

DeepDomain AI integrates job analysis, training paths, knowledge services, agent orchestration, and assertion-based sandbox execution. Automated tests verified principal service contracts and state transitions. Unknown sandbox tasks are not scored, model failures preserve rule-based results, deterministic scores resist prompt injection through free text, and stored paths and learner states can be retrieved consistently.

The offline evaluation provides limited but reproducible evidence. Dynamic ranking reached a 93.33% Top-3 hit rate and exceeded the static-

label baseline by 13.33 percentage points on 30 same-direction profiles. The threshold gate reproduced its own labels on 20 constructed samples. Because the profiles and gate labels were rule-derived, these results support functional correctness and offline ranking behavior rather than learner achievement, organizational benefit, or factual accuracy of generated resources.

6.2 Future Work

Repeated experiments should use fixed model versions, prompts, dependencies, and Docker image tags; their reports should include latency, cost, variance, and failure cases. Beyond these experimental controls, engineering work must handle concurrent sessions and adversarial sandbox submissions, compare model providers, audit bias, establish procedures for informed consent, enforce role-based access control, and define data-retention and deletion policies.

References

[1] Liu Y, Li W, Wang Y, et al. ISFJ-RAG: Interventional Suppression of Hallucinations via Counter-Factual Joint Decoding Retrieval-Augment Generation. Big Data and Cognitive Computing, 2026, 10(2): 56.

[2] Sun X, Hu Y, Gao X, et al. A cooperative multi-agent optimization approach for task offloading in vehicular edge computing systems. Discover Computing, 2025, 28(1): 348.

[3] Li H, Xing W, Li C, et al. I hear you: Enhancing middle school mathematics engagement and learning outcomes through teachable agent voice design. Computers & Education, 2026, 254: 105725.

[4] Saini G, Mishra A, Jadon S S.A neural-guided artificial bee colony with dynamic clustering and adaptive quality gating for global optimization. Expert Systems with Applications, 2026, 317: 131874.

[5] Ciroku F, de Berardinis J, Kim J, et al. RevOnt: Reverse engineering of competency questions from knowledge graphs via language models. Journal of Web Semantics, 2024, 82: 100822.

[6] Yang B, Shen Z. Knowledge graph construction and talent competency prediction for human resource management. Alexandria Engineering Journal, 2025, 121: 223-235.

[7] Ouared A, May M, Piau-Toffolon C, et al. TracePath: Modeling and Analyzing Competency Trajectories With Graph-Based Learning Analytics Over a Hybrid Polystore. Concurrency and Computation: Practice and Experience, 2026, 38(1): e70508.

[8] Li G, Yuan C, Kamarthi S, et al. Data science skills and domain knowledge requirements in the manufacturing industry: A gap analysis. Journal of Manufacturing Systems, 2021, 60: 692-706.

[9] Deshmukh P, Samrat H, Pervin N. Personalized learning path generation using large language models and sequential recommendations. Data & Knowledge Engineering, 2026, 164: 102595.

[10] Liu J, Yan C, Liu W, et al. Research on learner "emotion-behavior-ability" characteristics based on MOOC online education user profiles. Information Processing and Management, 2025, 62(3): 104026.

[11] Le H V, Ngo Q D. V-Sandbox for Dynamic Analysis IoT Botnet. IEEE ACCESS, 2020, 8: 145768-145786.

[12] Ivanchenko Y, Karpinski M, Ryzhakov M, et al.An Agent-Based Model of a Controlled Detonation System for Sandbox Analysis of Suspicious Software. Electronics, 2026, 15(11): 2348.

