ChatGPT

Scientifically Grounded Data Model for Employee Skills and Development

Introduction

Developing a robust data model for managing employee skills and development activities requires balancing practicality with adherence to established standards. In this report, we propose a **Django-friendly** data model grounded in academic and industry frameworks. The model centers on core entities like **Employee**, **Skill** (Competency), **Certification**, **DevelopmentAction**, **Project**, and **Budget**, with relationships designed to support AI-driven analytics while preserving employee anonymity. Our approach draws on well-regarded competency frameworks – notably the European e-Competence Framework (e-CF) and IEEE learning technology standards – to ensure a **common language** for skills and clear separation of personal data to mitigate bias [1] [2]. The following sections detail each model's fields and justification, referencing relevant literature and standards.

Standards for Skills & Competencies

Competency Frameworks: The data model aligns with established competency definitions to ensure interoperability and clarity. The **European e-Competence Framework (e-CF)**, for example, defines a set of 40 ICT competences with standardized descriptions and proficiency levels, creating a *"common language for competences, skills and proficiency levels across Europe"* [1]. Similarly, IEEE's competency standards (e.g., IEEE 1484 series) define *"competency" broadly as any skill, knowledge, ability, attitude, or learning outcome* [3]. A key principle from these frameworks is to treat each skill or competency as a **reusable, context-independent definition** [4]. This means our **Skill** model will represent the semantic definition of a skill independent of any particular employee or role, following the Reusable Competency Definition concept (RCD) from IEEE and IMS standards [4]. This approach allows skills to be consistently referenced across employees, projects, and training content.

Learning & Certification Standards: For development activities and certifications, we incorporate ideas from learning technology standards. For instance, the IMS Open Badges standard (now an ISO/IEC 23988 and W3C Verifiable Credential framework) treats certifications as portable credentials with metadata like issuer, issue date, and evidence. Each awarded certificate is an *assertion* with fields such as **issuer**, **date of achievement**, **expiration**, and evidence supporting the achievement [5]. Our model will reflect this by having a standalone **Certification** entity and linking employees to certifications with relevant metadata (issue dates, etc.). Development activities (training courses, mentorship, etc.) are modeled as **DevelopmentAction** records, aligning with IEEE and ISO learning experience standards by capturing the type of learning activity and its outcomes.

Privacy and Bias Considerations

To support future AI integrations **while preventing demographic bias**, the model strictly separates personal identifiable information from skill and performance data. Research on AI in hiring and development has shown that removing personal details (like name, gender, age, ethnicity) during evaluation can significantly reduce bias [2]. In our design, the **Employee** model contains only minimal necessary personal info (e.g., name, department, job title) and uses surrogate keys (IDs) to link to skills

and other records. These internal IDs (which can be UUIDs or numeric IDs) serve as **anonymous references** when exporting data to AI systems, so algorithms train on skill profiles and achievements without direct access to sensitive demographics. This "blind" data approach is akin to *blind hiring techniques* that focus on qualifications and skills over personal identifiers [2]. The model stores no attributes like age, gender, or ethnicity – ensuring compliance with privacy standards (e.g., GDPR) and reducing the risk of biased AI outcomes.

Core Data Model Overview

The core entities of the system and their relationships are illustrated below. We present each model with its key fields and relationships in tables for clarity. In summary:

- **Employee:** Basic employee info (minimal PII) and links to their skills, certifications, and development actions.
- **Skill (Competency):** A standalone definition of a skill or competence, possibly categorized and leveled, not tied to any single employee.
- **EmployeeSkill:** An association between an Employee and a Skill, including the employee's proficiency level and related metadata.
- **Certification:** A credential or certification object (e.g. professional certificate), defined independently and linked to employees when attained.
- **DevelopmentAction:** A record of a development activity (training, mentoring session, course, etc.), which can be tied to employees, skills, projects, and budgets.
- **Project:** An external or higher-level entity representing projects; included here to illustrate linking employees or skills to project contexts.
- **Budget:** Another external entity representing budget allocations, used to link funding information for development actions or projects.

These entities are interrelated: Employees possess skills (with proficiency), attain certifications, and engage in development actions. Projects can involve employees and require certain skills, while budgets may fund development actions. The relationships are designed to be explicit and traceable, which is useful for both REST API consumers and admin interface management.

Employee

The **Employee** model contains minimal personal data and primarily serves as a reference point for skills and activities. By keeping this model lean, we make it easier to anonymize skill data and avoid any attributes that could introduce bias. Each employee is identified by an ID and basic professional info like department and role.

Field	Type	Description
`id`	Auto/ UUID	**Primary Key** – Unique employee identifier (used as anonymous reference key in data).
`name`	String	Employee's name (could be real name or an alias code for privacy).
`department`	String	Department or team (e.g. "Engineering", "HR").
`job_title`	String	Job title or position (e.g. "Software Engineer").

Field	Type	Description
Relationships	–	– **Skills:** Many-to-many via **EmployeeSkill** (an employee can have multiple skills with proficiency ratings). – **Certifications:** Many-to-many via an Employee-Certification link (employee can hold multiple certs). – **DevelopmentActions:** One-to-many (one employee can have many development action records). – **Projects:** Many-to-many via a Project assignment link (if modeling project membership).

Rationale: We limit Employee fields to non-sensitive attributes (no birth date, gender, etc.), supporting bias mitigation as recommended in AI ethics research [2]. The use of a surrogate `id` (rather than, say, an email) ensures that all linkages to skills or activities use an **opaque key**. This design follows privacy-by-design principles and allows the system to supply AI models with **only anonymized identifiers** and skill data. The relationships defined ensure that from an Employee record we can navigate to all relevant competencies, training activities, and achievements for that person, which is important for building a skill profile in the admin UI or API.

Skill (Competency)

The **Skill** model represents a competency or skill definition. Each Skill is defined independent of any employee – mirroring the *"atomistic reusable competency definitions"* concept from HR-XML/IEEE standards [4]. A skill can be something like "Python Programming" or "Project Management". We may also include classification data to group skills (e.g., categories or frameworks like e-CF areas).

Field	Type	Description
`id`	Auto/ UUID	**Primary Key** – Unique identifier for the skill (could be a UUID or numeric ID; can serve as an external reference).
`name`	String	Name of the skill/competence (e.g., "Python Programming").
`description`	Text	Detailed description of what the skill entails.
`category`	String	(Optional) Category or domain (e.g., "Technical", "Soft Skill", or e-CF area like "Developer Skills").
`framework_ref`	String/ URI	(Optional) Reference code to an external framework or taxonomy (e.g., an e-CF competence code or HR-XML ID) for standard alignment.
`levels`	Enum (list)	(Optional) Applicable proficiency levels for this skill (e.g., levels 1–5 aligning to e-CF/EQF, or descriptors like Beginner/Expert).
Relationships	–	– **Employees:** Many-to-many via **EmployeeSkill** (skill can be possessed by many employees, at various proficiency levels). – **Certifications:** (Optional) Many-to-many (a skill can be linked to certifications that demonstrate it). – **Projects:** (Optional) Many-to-many (if modeling skill requirements per project). – **DevelopmentActions:** One-to-many (a training or course may target a specific skill).

Rationale: By separating Skill definitions from employees, we ensure consistency and reusability. This follows best practices from competency frameworks: *"represent the semantics of an individual competency,*

independently of its use in any particular context" [4]. The fields allow rich descriptions and classification. For example, `framework_ref` could store an official code (like an e-CF competence ID), anchoring the skill in a known structure – useful for reporting or AI models to leverage external competency ontologies. We include an optional list of `levels` to describe how proficiency can be measured for this skill. (The actual level an employee has is stored in the **EmployeeSkill** junction, not in the Skill itself.) This design is consistent with standards like e-CF which link competences with defined proficiency levels [6]. It's also similar to the HR-XML approach where a competency definition is separate and proficiency ratings are attached when profiling an individual [4] [7].

EmployeeSkill (Proficiency Profile)

EmployeeSkill is an association model capturing an individual employee's proficiency in a given skill. This implements the many-to-many relationship between Employees and Skills, with additional data (proficiency level, evidence, etc.). Each record can be seen as one entry in an employee's skill **profile** or competency matrix.

Field	Type	Description
`id`	Auto	Primary key (if using an explicit model for the M2M relation).
`employee`	FK→Employee	**Foreign Key** – The employee who has this skill.
`skill`	FK→Skill	**Foreign Key** – The skill which the employee has.
`proficiency_level`	Integer/Enum	Proficiency rating for this employee in this skill. Could be an integer scale (e.g., 1–5) or an enum ("Beginner", "Intermediate", "Expert"), potentially aligned with e-CF level definitions.
`last_updated`	Date	Date when this proficiency was last assessed/updated.
`evidence`	Text/URL	(Optional) Notes or link to evidence (e.g., project work, assessment results) supporting this proficiency.

Rationale: Storing proficiency at the intersection of Employee and Skill is crucial. According to HR-XML standards, competency profiles often include a *competency weight or level* to quantify an individual's skill, along with evidence for that assessment [7]. This model follows that practice: the `proficiency_level` field corresponds to a skill rating or weight. By having an explicit model, we can easily extend it to store additional context such as who assessed the skill or what evidence was used (certificates, project outcomes, etc.). For instance, if an employee improves their skill via training, the admin can update the proficiency level and attach a note or link (perhaps to a training record or certificate). This separation also keeps the core Skill definition free of person-specific data. From a REST API perspective, this model allows queries like "get all skills and levels for employee X" or "list all employees who have skill Y above proficiency 3," which are common in talent management use cases. In the Django admin, **EmployeeSkill** could be managed via inline forms on the Employee page for convenient editing of an employee's skill set.

Certification

The **Certification** model represents official certifications, qualifications, or credentials that are relevant to employee skills. Examples might include certifications like "AWS Certified Solutions Architect" or "PMP (Project Management Professional)". We treat certifications as first-class entities, separate from

employees, so that they can be linked and referenced independently (and possibly reused if multiple employees hold the same cert).

Field	Type	Description
`id`	Auto/ UUID	**Primary Key** – Unique identifier for the certification.
`name`	String	Name of the certification (e.g., "AWS Solutions Architect Associate").
`issuer`	String	Issuing organization or authority (e.g., "Amazon Web Services").
`description`	Text	Description of the certification (what it certifies, scope, etc.).
`validity_period`	String/ Interval	(Optional) Duration the certification is valid for (e.g., "3 years") if it expires.
`external_ref`	String/ URL	(Optional) External reference or URL (link to the credential's details or an Open Badge JSON).
Relationships	–	– **Employees:** Many-to-many via an association (each Certification can be held by many Employees; an Employee can have many Certifications). – **Skills:** Many-to-many (optionally, link a certification to the Skills it validates; e.g., "AWS Architect" cert might relate to cloud computing skills).

Rationale: Certifications are modeled separately to reflect their role as verifiable achievements. By not embedding certification info directly in Employee, we avoid data redundancy and allow rich metadata on each certificate. This design echoes the Open Badges concept where an *Achievement* (Badge) is defined with its own properties and then *awarded* to individuals [5]. Key metadata like `issuer` and `validity_period` are included because they are often needed in HR systems (for example, to remind when a certification needs renewal). The Employee–Certification relationship would likely be implemented via a join table (which can include additional fields such as the date of attainment, certificate ID, etc.). In many cases, we would add an intermediary model like **EmployeeCertification** to capture details of when the employee obtained the cert, score, etc., but from a high-level model perspective it's a many-to-many link. We also note a potential relationship to Skills: while not strictly required, it can be useful to tag which skills a certification covers. This can help in queries like "who has certifications that prove skill X," and is in line with competency frameworks linking qualifications to competencies (e.g., e-CF dimension 4 provides examples of knowledge/skills related to a competence).

DevelopmentAction

The **DevelopmentAction** model records any developmental activity an employee undertakes: training sessions, courses (online or offline), mentoring engagements, workshops, etc. Each record captures what the activity was, who was involved, and when, and it can optionally link to targeted skills, projects, and budget sources.

Field	Type	Description
`id`	Auto	**Primary Key** – Unique identifier for the development action.

Field	Type	Description
`type`	Enum/String	Type of development action (e.g., `"Training"`, `"Course"`, `"Mentoring"`). This can be an enumeration of the allowed types.
`title`	String	Title or name of the activity (e.g., "Django REST Framework Workshop").
`description`	Text	Description or notes about the activity (content, outcomes, etc.).
`date`	Date	Date when the activity took place (or started).
`duration`	String/Int	(Optional) Duration of the activity (e.g., "3 days" or number of hours).
`employee`	FK→Employee	The employee who underwent this development action. *(If multiple employees can attend one action, this could be a many-to-many, but typically we record individual participation or use separate records per employee for group trainings.)*
`skill`	FK→Skill	(Optional) The primary skill/competency targeted or improved by this activity.
`certification`	FK→Certification	(Optional) Certification awarded by this action (if this training leads to a cert or is a cert program).
`project`	FK→Project	(Optional) Link to a Project context – e.g., the training is part of Project X's initiative.
`budget`	FK→Budget	(Optional) Link to a Budget source – e.g., funded by Budget Y.

Rationale: Capturing development activities in a structured way is key for both employee development tracking and feeding data to AI systems that might recommend learning or measure growth. By classifying the `type` of action, we can distinguish how skills are developed (formal course vs. on-the-job training vs. mentorship). The optional links to **Skill**, **Project**, and **Budget** make the model flexible: for instance, a record might show *Employee A attended "Advanced Python Training" (type=Course) on 2025-05-01, targeting Skill=Python Programming, related to Project=CloudMigration, funded by Budget=DeptTrainingQ2*. Such linkages support multi-dimensional analysis (e.g., evaluating ROI of training by project or budget, or listing all trainings related to a certain skill). In the Django admin, this model can be made accessible so admins or managers can log new activities. For the REST API, endpoints like `/development-actions/` allow retrieval of an employee's development history or all activities for a given skill, etc.

From a standards perspective, this model can also be mapped to learning experience standards. For example, an entry here is analogous to a learning record; if needed, the system could export these as xAPI statements or similar. The design ensures that **development data remains separate from personal data**, aligning with the principle of keeping the skill-learning ecosystem anonymized (the employee link is just an ID). Furthermore, having clear foreign keys for project and budget meets the requirement of mapping to those external objects, while keeping the core focus on development.

Project

The **Project** model represents projects or initiatives in the organization that may be related to skill utilization or development. In many organizations, projects have specific skill needs and involve certain employees. While project management itself might be handled in another system, we include a simple Project model to illustrate how it can link with the skills framework.

Field	Type	Description
`id`	Auto/ UUID	**Primary Key** – Unique project identifier.
`name`	String	Project name (e.g., "AI Platform Development").
`description`	Text	Project description or scope.
`start_date`	Date	(Optional) Start date of the project.
`end_date`	Date	(Optional) End date or deadline of the project.
Relationships	–	– **Employees:** Many-to-many (a project involves multiple employees; can be modeled via an assignment or membership table). – **Skills:** Many-to-many (optional, to list required or involved skills in the project). – **DevelopmentActions:** One-to-many (many development actions can be linked to a project as context). – **Budgets:** Many-to-one or one-to-one (a project may have one budget associated).

Rationale: Including projects provides context for why certain skills are important or developed. For example, if Project X demands a rare skill, the company can plan development actions for employees to acquire that skill. Our model makes it possible to query, say, “which projects are linked to cybersecurity skills and who on those projects has relevant training?”. By keeping Project minimal here, we allow integration with a larger project management system. The **Skill <-> Project** relationship (if used) essentially creates a skill requirement matrix for projects, which is a known practice in resource management. Projects can also have a relationship with **Budget** (each project might have a budget or multiple budgets, but that can be simplified as needed). From an admin perspective, the Project model can be managed or imported, and its connection to employees (assignments) can be handled via Django’s many-to-many relations or a separate model like ProjectAssignment.

Budget

The **Budget** model represents a funding source or allocation, particularly for development and training purposes. In an HR development context, budgets might be allocated per department, project, or program and we may want to track how much of a budget is spent on certain training.

Field	Type	Description
`id`	Auto/ UUID	**Primary Key** – Unique budget identifier.
`name`	String	Budget name or code (e.g., "FY2025 L&D Budget", "Project X Training Fund").

Field	Type	Description
`amount`	Decimal	(Optional) Total amount of the budget (in relevant currency).
`description`	Text	(Optional) Description of the budget's purpose.
`start_date`	Date	(Optional) Start of budget period.
`end_date`	Date	(Optional) End of budget period.
Relationships	–	– **Projects:** One-to-many (one budget can fund multiple projects, or one project may draw from one budget; depends on use-case). – **DevelopmentActions:** One-to-many (multiple development actions can be funded by the same budget source).

Rationale: The Budget model is included to fulfill the requirement of mapping to budget objects and to facilitate financial tracking of development activities. By linking Budget to DevelopmentAction, we could calculate how much budget is used for training each employee or skill. While the exact relationship between projects and budgets can vary (some organizations have per-project budgets, others have central L&D budgets), the model is flexible to accommodate either. The presence of Budget as a separate entity also underscores a separation of concerns: financial data is tracked in its table, and only references (foreign keys) appear in the development or project records. This keeps the core skill data model focused, but extensible. In a REST API context, one could filter development actions by budget to produce a report of spending, etc. From an admin standpoint, budget records can be managed and then selected when logging a development action, ensuring consistency in how funding is recorded.

Relationships and Integrity Considerations

All the above models are connected via foreign keys or many-to-many relations, ensuring referential integrity and enabling rich querying: - **EmployeeSkill** enforces that any skill rating is tied to a valid Employee and Skill. If an Employee is removed or a Skill deleted, cascade or careful handling ensures no orphaned proficiency records. - **Employee–Certification** linking (via an implicit or explicit join model) means an Employee can't have a certification that isn't defined in the Certification table, and vice versa. - **DevelopmentAction** ties together multiple entities. We allow nullable foreign keys for Skill, Project, Budget, etc., because not every training will be linked to all of those – but when they are, it must be a valid reference. For example, `project_id` in a DevelopmentAction must correspond to a Project that exists. This design also allows a development action to stand alone (not part of a project or budget) if needed. - **Project** and **Budget** themselves might be external or higher-level entities. If they are managed outside of this app, we could store just an external key or integrate via Django's contenttypes. However, given they are included as models here, we assume they are managed within the same system for simplicity.

Notably, by using **IDs and foreign keys** for all relationships, we facilitate anonymity and future AI use: an AI system can be given data like *Employee #123 has Skill #45 (Proficiency 4)*, *Employee #123 took DevelopmentAction #789 targeting Skill #45*, etc., without ever seeing personal identifying info. This approach addresses the need for "clear, anonymous keys" as the basis of data exchange, preventing any AI from inadvertently learning protected attributes. It essentially implements a form of **blind data modeling**, as advocated in bias mitigation research [2].

Furthermore, the model can support administrative and analytical needs. For instance: - **Django Admin:** We can register these models so that HR managers can input and update data through admin forms. The relations (e.g., EmployeeSkill inline on Employee, or selecting Skills in a DevelopmentAction form)

make data entry intuitive. - **REST API Endpoints:** Each model would typically have its own endpoint (`/employees/`, `/skills/`, etc.), and we can provide nested or filterable endpoints (e.g., `/employees/{id}/skills/` to get an employee's skill profile). The clear separation also means we can impose different permissions – e.g., personal data (Employee info) could be more restricted than aggregated skill data.

Scientific Basis for Modeling Choices

Each element of the data model is backed by concepts from literature or standards: - **Competency Separation:** The separation of Skill definitions from person-specific proficiency draws on IEEE & IMS specifications for competencies. As noted, *"an individual competency [should be represented] independently of a person or context,"* enabling reuse across systems [4]. This is a cornerstone of frameworks like IMS RDCEO and IEEE RCD. Our Skill and EmployeeSkill split realizes this principle in practice. - **Proficiency Levels:** We include proficiency as a measurable attribute (rather than a binary have/doesn't have skill). This is supported by competency modeling research and standards. For example, HR-XML provides a *CompetencyWeight* to express proficiency and even suggests recording evidence for how the competency is evaluated [7]. Likewise, e-CF links each competence to proficiency levels (e.g., Level 1–5) [1]. Our model's proficiency field and evidence field align with these ideas, ensuring we can capture not just that "Employee knows Skill X" but *how well* or *in what capacity* they know it. - **Certifications as Credentials:** By modeling certifications separately, we align with educational data standards that treat credentials as verifiable, shareable records. The Open Badges specification, for instance, defines that a badge (certification) contains metadata like issuer, date, etc., and is awarded to a person [5]. We mimic that structure, which not only is logical for our application but also means in the future we could integrate with digital credential platforms (issuing or importing badges). - **Learning & Development Tracking:** The inclusion of **DevelopmentAction** resonates with the concept of learning histories or learning experience tracking in talent management. While not directly from one standard, it is informed by best practices in corporate L&D that emphasize recording training interventions and linking them to competencies (often called a Learning Record Store in xAPI terms). This helps in analyzing the efficacy of training and is a dataset that AI could use (e.g., to recommend next trainings or to predict project readiness based on past training). - **Privacy by Design:** Finally, the conscious avoidance of storing demographic data in the skill system is backed by numerous studies highlighting bias in AI due to such data [2]. By only storing job-relevant information (skills, training, performance indicators) and keeping personal info minimal and isolated, we follow the guidance from ethical AI research that recommends anonymization and focus on skills to promote fairness [2].

Future Considerations for AI Integration

Looking ahead, this data model sets a foundation for AI-powered features. With clearly defined models, one could implement: - **Skill Gap Analysis:** Comparing required project skills to the team's EmployeeSkill profiles to identify gaps. - **Learning Recommendations:** Using DevelopmentAction history and Skill targets to recommend courses (AI can use the structure to find what skills an employee lacks relative to their role or career path). - **Succession Planning:** Because skills and certifications are well-structured, queries for who meets certain criteria are straightforward (e.g., find employees with Skill=A at proficiency >=4 and Certification in that area for a new project role). - **Bias Auditing:** Since employee demographic data isn't entangled with skills, any AI model's decisions (e.g., recommending someone for a role based on skills) can be more easily audited for bias – it would be based purely on the merit data available. If needed, sensitive data could be introduced in a controlled way to check for fairness, but the default state is anonymized.

The REST API can expose these models in a RESTful way, and because the data model adheres to common concepts, integrating external AI or analytics systems (via JSON/CSV export or direct API calls) is facilitated. Each competency or certification can be referenced by a stable ID that could correspond to external frameworks (e.g., if using a global skill ontology, the `framework_ref` in Skill could store that). This opens the door to interoperability, such as aligning the internal skills with external labor market taxonomies (like ESCO or O*NET), which is often beneficial in talent management systems [6].

Conclusion

The proposed data model provides a comprehensive yet clean structure for managing employee skills, development activities, certifications, and their linkage to organizational constructs like projects and budgets. By grounding the design in respected frameworks (e-CF, IEEE 1484, HR-XML) and academic insights, we ensure that the model is not only normalized and logical but also future-proof and unbiased. Each model in the schema has a clear purpose and scientific justification – from treating skills as reusable competencies, to capturing proficiency levels with evidence, to isolating personal data to prevent AI bias. This structured approach will support robust Django REST API development (with intuitive endpoints and admin interfaces) and lays the groundwork for advanced AI-driven functionalities in a fair and ethical manner.

Sources:

- European e-Competence Framework – common language for competences and proficiency levels [1].
- IEEE P1484.20.3 – competency defined as skill, knowledge, ability, etc., for learning systems [3].
- HR-XML / IEEE RCD – competencies defined independent of context (reusable competency definitions) [4].
- HR-XML Competencies – proficiency level via competency weight, with evidence for evaluation [7].
- Open Badges (IMS) – certifications carry issuer, date, and evidence metadata as verifiable achievements [5].
- Bias in AI HR systems – anonymizing personal data (names, gender, etc.) focuses assessment on skills and reduces bias [2].

---

[1] [6] European e-Competence Framework (e-CF) | European Skills, Competences, Qualifications and Occupations (ESCO)
https://esco.ec.europa.eu/en/about-esco/escopedia/escopedia/european-e-competence-framework-e-cf

[2] A Comprehensive Review of AI Techniques for Addressing Algorithmic Bias in Job Hiring
https://www.mdpi.com/2673-2688/5/1/19

[3] IEEE Sharable Competency Definitions (P1484.20.3) | ADL Initiative
https://adlnet.gov/working-groups/ieee-sharable-compentency-definitions/

[4] HR-XML Competency - InLOC
https://www.simongrant.org/InLOC/HR-XML%2BCompetency

[5] Open Badges Specification | IMS Global Learning Consortium
https://www.imsglobal.org/spec/ob/v3p0

[7] HR-XML Competencies Schema | Download Scientific Diagram
https://www.researchgate.net/figure/HR-XML-Competencies-Schema_fig1_262312184