ChatGPT

Education Domain Ontology (Claude Code Ready)

This ontology provides a structured **education domain model** with all fundamental components. It is formatted in **YAML within Markdown** for easy integration with Claude Code v2.1.14. The ontology defines **Object Types** (entities like students, courses), their **Properties**, **Link Types** (relationships between entities), and **Action Types** (workflows/interventions in the domain). This structure helps Claude Code reason about educational concepts for code suggestions, data generation, and natural language understanding. (**Note:** In Claude Code, you can include this ontology in a `CLAUDE.md` file at your project root so it's automatically pulled into context [1].)

Object Types (Entities)

Entities are the core objects in the education domain (students, instructors, courses, etc.). Each object type includes its key properties and the relationships it participates in.

```yaml
object_types:
  - name: Student
    description: A learner enrolled in the educational program.
    properties:
      - name: student_id
        type: string
        description: Unique identifier for a student.
      - name: name
        type: string
        description: Full name of the student.
      - name: email
        type: string
        description: Contact email address of the student.
      - name: enrolledCourses
        type: list<Course>
        description: Courses the student is currently enrolled in.
      - name: completedCourses
        type: list<Course>
        description: Courses the student has completed.
    relationships:
      - type: enrolled_in   # Student -> Course
        target: Course
        description: Student is **enrolled in** a Course.
      - type: completed_course   # Student -> Course
        target: Course
        description: Student has **completed** (passed) a Course.
      - type: has_submission   # Student -> AssignmentSubmission
        target: AssignmentSubmission
        description: Student has a submission for an assignment/assessment.

  - name: Instructor
```

```
  description: An educator who teaches courses.
  properties:
    - name: instructor_id
      type: string
      description: Unique identifier for an instructor.
    - name: name
      type: string
      description: Full name of the instructor.
    - name: email
      type: string
      description: Contact email address of the instructor.
    - name: coursesTaught
      type: list<Course>
      description: Courses this instructor teaches.
  relationships:
    - type: teaches    # Instructor -> Course
      target: Course
      description: Instructor **teaches** a Course (inverse of Course.taught_by).

- name: Course
  description: A course of study on a particular subject.
  properties:
    - name: course_id
      type: string
      description: Unique course code or identifier.
    - name: title
      type: string
      description: Title of the course.
    - name: description
      type: string
      description: Summary of the course content.
    - name: credits
      type: integer
      description: Credit value or weight of the course.
    - name: instructor
      type: Instructor
      description: The Instructor leading/teaching this course.
    - name: students
      type: list<Student>
      description: Students currently enrolled in this course.
    - name: lessons
      type: list<Lesson>
      description: Lessons or units that make up the course.
    - name: assessments
      type: list<Assessment>
      description: Assessments (assignments, quizzes, exams) in the course.
  relationships:
    - type: taught_by   # Course -> Instructor
      target: Instructor
      description: Course is **taught by** an Instructor.
    - type: has_prerequisite   # Course -> Course
```

```
    target: Course
    description: Course has another Course as a prerequisite.
  - type: has_assessment   # Course -> Assessment
    target: Assessment
    description: Course includes an Assessment (assignment/quiz/exam).

- name: Lesson
  description: A unit of teaching (lesson or module) within a course.
  properties:
   - name: lesson_id
     type: string
     description: Unique identifier for the lesson.
   - name: title
     type: string
     description: Title or topic of the lesson.
   - name: content
     type: string
     description: Instructional content of the lesson (text, video, etc.).
   - name: course
     type: Course
     description: The Course this lesson belongs to.
  # (Relationships for Lesson are typically via Course; e.g., Course -> Lesson list)

- name: Assessment
  description: A test or assignment to evaluate learning (could be an assignment, quiz, or exam).
  properties:
   - name: assessment_id
     type: string
     description: Unique identifier for the assessment.
   - name: title
     type: string
     description: Title or name of the assessment.
   - name: category
     type: string
     description: Type/category (e.g. "assignment", "quiz", "exam").
   - name: max_score
     type: number
     description: Maximum achievable score.
   - name: due_date
     type: date
     description: Due date or deadline (if applicable).
   - name: course
     type: Course
     description: The Course this assessment is part of.
  relationships:
   - type: part_of_course   # Assessment -> Course
     target: Course
     description: (Implicit via Course.assessments; same as Course has_assessment)

- name: AssignmentSubmission
  description: A student's submission for an assignment or assessment.
```

```
  properties:
    - name: submission_id
      type: string
      description: Unique identifier for the submission.
    - name: content
      type: string
      description: The work submitted (text, file link, etc.).
    - name: timestamp
      type: datetime
      description: Time the submission was made.
    - name: student
      type: Student
      description: The Student who made the submission.
    - name: assessment
      type: Assessment
      description: The Assessment this submission is for.
    - name: grade
      type: Grade (optional)
      description: Grade/score given for this submission (once graded).
  relationships:
    - type: submitted_by   # AssignmentSubmission -> Student
      target: Student
      description: The submission was **submitted by** a Student.
    - type: for_assessment   # AssignmentSubmission -> Assessment
      target: Assessment
      description: The submission is for a specific Assessment.

- name: Grade
  description: A grade or score awarded for a submission or course performance.
  properties:
    - name: grade_id
      type: string
      description: Unique identifier for the grade record.
    - name: value
      type: number
      description: Numeric grade or score value.
    - name: letter
      type: string (optional)
      description: Letter grade or qualitative grade (optional, e.g. "A", "Pass").
    - name: feedback
      type: string (optional)
      description: Feedback comments from the grader.
    - name: student
      type: Student
      description: The Student who received this grade.
    - name: assessment
      type: Assessment
      description: The Assessment for which this grade was given.
    - name: grader
      type: Instructor
      description: The Instructor or grader who assigned this grade.
```

```
relationships:
 - type: grade_for  # Grade -> AssignmentSubmission/Assessment
   target: AssignmentSubmission
   description: The grade is for a particular submission (links to submission).
 - type: graded_by  # Grade -> Instructor
   target: Instructor
   description: The grade was **graded by** an Instructor.
```

Usage notes: Object type definitions are kept **concise and normalized**. Each entity has an `..._id` primary identifier and references to related entities by type (e.g. `Course.instructor` of type Instructor, `Assessment.course` of type Course). Lists (e.g. `list<Student>`) indicate one-to-many relationships. Descriptions are included to help both humans and Claude understand each field's meaning. The above structure is **machine-readable** by Claude (the YAML syntax makes it easy for the model to parse and follow the schema) while also being human-readable in Markdown.

Link Types (Relationships)

Link Types define how entities relate to each other. They are essentially the named relationships (edges) between object types. Below is a list of relationship types, specifying their domain (source), range (target), and meaning. (Some correspond to common education relationships like **teaching** and **enrollment**, which we see in practice [2].)

```
link_types:
 - name: enrolled_in
   domain: Student
   range: Course
   description: A Student is **enrolled in** a Course (the student takes/attends the course).
 - name: teaches
   domain: Instructor
   range: Course
   description: An Instructor **teaches** a Course (inverse of "taught_by" on Course).
 - name: taught_by
   domain: Course
   range: Instructor
   description: A Course is **taught by** an Instructor (inverse of "teaches").
 - name: has_prerequisite
   domain: Course
   range: Course
   description: Course A requires completion of Course B first (prerequisite relationship).
 - name: has_assessment
   domain: Course
   range: Assessment
   description: A Course **has** an Assessment (part of the course's assessments).
 - name: has_submission
   domain: Student
   range: AssignmentSubmission
   description: A Student **has** an AssignmentSubmission for an assessment.
 - name: submitted_by
   domain: AssignmentSubmission
```

```
  range: Student
  description: An AssignmentSubmission was **submitted by** a Student.
 - name: for_assessment
  domain: AssignmentSubmission
  range: Assessment
  description: An AssignmentSubmission is **for** a particular Assessment.
 - name: graded_by
  domain: Grade
  range: Instructor
  description: A Grade was **graded by** an Instructor.
 - name: grade_for
  domain: Grade
  range: AssignmentSubmission
  description: A Grade is **for** a particular AssignmentSubmission (assessment attempt).
 - name: completed_course
  domain: Student
  range: Course
  description: A Student has **completed** a Course (finished with passing grade).
```

Usage notes: These links often appear as fields in the object definitions (for example, `enrolled_in` is implied by the `Course.students` list and `Student.enrolledCourses`). We list them explicitly for clarity and to ensure the ontology includes all relationship semantics. Tools or code can use these definitions for graph queries or validation (e.g., preventing a student from enrolling twice). In Claude's context, naming these link types explicitly helps it understand queries like "Who teaches course X?" or "List students enrolled in course Y" by mapping to the relevant relationships.

Action Types (Workflows)

Action Types represent key **interventions or workflows** in the education domain – basically, processes or events that change state or involve multiple entities. They often correspond to **common tasks** (enrolling in a course, submitting an assignment, grading, etc.) and will typically produce or update the above entities/relationships. We define each action with its name, a description, expected inputs, and outcomes (including any events triggered).

```
action_types:
 - name: EnrollStudentInCourse
  description: Enroll a student into a course.
  inputs:
   - student_id: ID of the Student to enroll
   - course_id: ID of the Course to enroll into
  outcome:
   - Adds the Student to the Course.students list (creates Student–Course enrolled_in link).
   - Triggers an event **course_enrolled** (actor=Student, target=Course, timestamp).
 - name: DropStudentFromCourse
  description: Remove a student from a course (unenroll).
  inputs:
   - student_id: ID of the Student
   - course_id: ID of the Course
  outcome:
```

```
    - Removes the Student's enrollment (deletes the enrolled_in relationship or marks it inactive).
    - (Optional) Triggers an event **course_dropped** (if tracking drops).
  - name: SubmitAssignment
    description: Student submits work for an assignment/assessment.
    inputs:
      - student_id: ID of the Student
      - assessment_id: ID of the Assessment (e.g. assignment or quiz) being submitted
      - content: The submission content (answers, file link, etc.)
    outcome:
      - Creates an **AssignmentSubmission** object linking the Student and Assessment (stores content and timestamp).
      - Triggers an event **assessment_submitted** (or **quiz_submitted** for quizzes) with details [3].
  - name: GradeSubmission
    description: Instructor assigns a grade to a student's submission.
    inputs:
      - submission_id: ID of the AssignmentSubmission to grade
      - grade_value: Score or grade value to assign
      - feedback: Optional feedback text
    outcome:
      - Creates a **Grade** record linked to the Student (via submission -> student) and the Assessment.
      - Attaches the Grade to the AssignmentSubmission (AssignmentSubmission.grade).
      - Triggers an event **grade_assigned** (to log that grading occurred; could also notify student).
  - name: CompleteLesson
    description: Mark a lesson as completed by a student.
    inputs:
      - student_id: ID of the Student
      - lesson_id: ID of the Lesson completed
    outcome:
      - Records the completion (could update a Student progress record or generate a completion entry).
      - Triggers an event **lesson_completed** (actor=Student, target=Lesson) for analytics/progress tracking [3].
  - name: CreateCourse
    description: Create a new course in the catalog.
    inputs:
      - title: Course title
      - description: Course description
      - instructor_id: ID of Instructor who will teach the course
      - credits: Credit value for the course (optional)
    outcome:
      - Creates a new **Course** object with given details (and sets up the taught_by link to the Instructor).
      - Triggers an event **course_created** (with metadata like who created it, timestamp).
  - name: AddAssessment
    description: Add a new assessment (assignment/quiz) to a course.
    inputs:
      - course_id: ID of the Course
      - title: Name of the assessment
      - category: Type of assessment ("assignment", "quiz", etc.)
```

```
    - max_score: Maximum score
    - due_date: Due date (if applicable)
  outcome:
    - Creates a new **Assessment** object and links it to the Course (has_assessment relationship).
    - (Optionally triggers an event **assessment_created** for audit trail).
 - name: AwardCertificate
  description: Award a completion certificate to a student for finishing a course.
  inputs:
    - student_id: ID of the Student
    - course_id: ID of the Course completed
  outcome:
    - Creates a certificate record (not modeled above, but could be a new entity or a special Grade
record).
    - Triggers an event **certificate_earned** (actor=Student, target=Course) marking course
completion [3].
```

Usage notes: Action types serve as **templates for workflows**. They can guide how to implement features or processes in code. For example, an `EnrollStudentInCourse` function in code would take a student and course, then establish the appropriate data links and log an event. By defining these actions in the ontology, Claude can better plan multi-step processes (e.g., in an agentic workflow, the model might break down a user request into these high-level actions). Each action's **inputs** clarify what data is needed, and **outcome** explains what the action does (which entities/relationships it affects and which events it emits). This ensures consistency – e.g., any time an enrollment happens, the model knows to create the `enrolled_in` link and fire a `course_enrolled` event, rather than doing something ad-hoc. (In the ONE project, they similarly had events like `course_enrolled`, `lesson_completed`, `quiz_submitted` to log such actions [3].)

Integration & Best Practices Guide

To make the most of this ontology in your Claude Code workflow, consider the following best practices (updated for **Claude Code v2.1.14**):

- **Include in Context:** Place the ontology (or relevant parts of it) in your project's `CLAUDE.md` file or as a referenced knowledge file. Claude Code automatically loads `CLAUDE.md` content at session start [1], so by documenting your ontology there, you ensure the model always "knows" these domain concepts. Keep the formatting clean (as shown above) with Markdown headers and YAML blocks – this is optimal for Claude's parsing and reference.

- **Layer and Scope the Knowledge:** This ontology covers multiple layers of information – **entities (People/Things), relationships, actions, events** – akin to a multi-dimensional approach [4]. Ensure your workflow captures each layer. For instance, when a new feature is introduced (say, a new type of assessment), update the ontology rather than creating stray ad-hoc structures. This aligns with the best practice of mapping all features to the established ontology **instead of creating custom schema** [5]. It keeps Claude's understanding consistent and avoids confusion.

- **Use Ontology for NL Understanding:** Leverage the ontology to **interpret user prompts and plan tasks**. Claude can map natural language requests to ontology entities and actions. For example, if a user asks in natural language, "Enroll Alice in Math 101 and give her access to the first quiz," Claude (with this ontology in context) will recognize "enroll" relates to the

`EnrollStudentInCourse` action and “quiz” relates to an `Assessment` of category "quiz" in that course. The model can then generate the appropriate code or sequence of calls (e.g., calling an enrollment function, then perhaps creating a submission or unlocking a quiz) following the ontology’s structure. The ontology terms serve as a **shared vocabulary** between your prompts and Claude’s reasoning.

- **Guide Code Generation and Data Shaping:** When asking Claude to generate code (database schemas, class definitions, API endpoints, etc.) or data (like JSON examples), **refer to the ontology** so it uses the correct structure. For instance, if generating a database model for courses and enrollments, remind Claude to use the `Student`, `Course` tables and an `enrolled_in` linking table as per the ontology. Claude Code will use the ontology to fill in correct field names and relationships rather than inventing its own. This results in more **consistent, ontology-compliant code** (which you can even automate checks for – see next point). Similarly, for data generation, you can prompt Claude to output dummy data in YAML/JSON following the ontology schema (e.g., a list of Student objects with their properties), and it should adhere closely to the defined types and fields.

- **Ontology Validation Hooks:** Claude Code allows custom validation hooks or slash commands. You can set up an **ontology compliance check** (for example, a `/validate` command or a git hook) that inspects code changes to ensure they don’t violate the ontology – e.g. no undefined fields or extra tables outside the ontology [5] [6]. In v2.1.14, hooks and commands are robust; you could write a script that reads the YAML ontology and verifies your code (or database schema migrations) against it. This way, whenever you or Claude generate new code, you can quickly catch if it strays from the defined model (as was done in one case where a hook prevented creating a custom “videos” table and suggested using an existing ontology entity instead [7]).

- **Retrieval-Augmented Generation (RAG) Integration:** If your application uses a knowledge base or document store (e.g., course materials, student records), combine it with the ontology for more precise RAG. The ontology gives structure to queries – for example, an agent can form a query like “fetch all `Lesson` content for Course X” or “retrieve past `Grade` records for Student Y”. By using ontology terms, you ensure the retrieval targets the right pieces of data. In Claude Code, you might have MCP tools (APIs, database connectors) that Claude can call; design those APIs around ontology concepts (e.g., an API endpoint `/courses/{id}/students` corresponding to the enrolled_in relationship). This makes it easier for Claude to choose the right tool. Claude Code v2.1.14 improved external tool usage and memory, so the model can maintain awareness of the ontology while pulling in fresh data. For example, with a vector search MCP tool, Claude could embed a query like “Course:title:Math 101 AND Assessment:category:quiz” to find relevant quiz content, guided by the ontology schema.

- **Maintain and Evolve Ontology with Claude:** As your project grows, update the ontology document and let Claude know about changes. You can even ask Claude (in a development session) to suggest ontology modifications when a new concept arises. For instance, if you introduce a new role “TeachingAssistant”, Claude can propose how to integrate it (maybe as another object type linked to Course and Instructor). This co-evolution keeps the model’s context aligned with the codebase’s reality. Always keep the ontology **concise and up-to-date** – Claude Code’s context window is large, but clarity helps it focus. Version 2.1.14 has improved handling for long contexts and history, but it’s wise to trim unnecessary details and use **summaries or modular files** if the ontology becomes very large. (You could break the ontology into sections – e.g., academic structure vs. user data – and load on demand, but for most education domains the above should be fine in one file.)

- **Use Ontology for Intelligent Suggestions:** Because the ontology adds semantic knowledge, Claude can do more than just autocomplete code. It can **reason** about the domain. For example, if you're writing a function to calculate a student's GPA, Claude knows from the ontology that relevant pieces are the Student's completedCourses and Grades. It might suggest iterating over `Student.completedCourses` and averaging the associated `Grade.value` fields. Such reasoning stems from the structured context you've given it. In essence, the ontology acts as a mini knowledge graph that Claude uses internally to understand context and maintain consistency.

By following these practices, you integrate the ontology deeply into your Claude Code workflow. This means fewer misunderstandings and faster development. Claude will use the ontology to interpret instructions correctly and produce coherent results. In Anthropic's internal usage, this kind of ontology-driven approach led to significant speed-ups and accuracy gains (e.g., **0 ontology violations after integration** and higher consistency in code style). Adopting a similar strategy in your single-developer codebase can turn Claude into a truly **grounded coding partner** rather than a generic autocomplete, as it will be operating with the clear rules and context of your education domain model.

---

1 Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

2 3 4 GitHub - one-ie/ontology: ONE Ontology
https://github.com/one-ie/ontology

5 6 7 How We Trained Claude Code to Build at Machine Speed
https://one.ie/news/claude-code-integration-complete/