perplexity

Based on this business Logic, write the database Schema and use prisma ORM. Determine if we will need a relational database as well, and the choice for this will be Postgres

Of course. Based on the established business logic for the Enterprise Agentic Workflow Engine (EAWE), here is the database schema design using Prisma ORM.

Database Strategy: A Hybrid Approach

To meet all requirements of the business logic, we will use a hybrid database strategy. A simple relational database alone is insufficient.

1. **Relational Database (PostgreSQL):** This will be our primary data store for all structured, transactional data. It will manage users, workflow definitions, execution states, audit logs, and HITL tasks. PostgreSQL is the ideal choice due to its robustness, transactional integrity (ACID compliance), and powerful features like JSONB support, which we will leverage heavily. Prisma ORM will be used to interact with this database.
2. **Vector Database (Qdrant):** This specialized database is required for the semantic search and Retrieval-Augmented Generation (RAG) capabilities [projects.ai_search]. It is designed for the sole purpose of storing high-dimensional vector embeddings and performing ultra-fast similarity searches with advanced filtering. Storing and querying vector embeddings in Postgres (via extensions like `pgvector`) is possible, but a dedicated solution like Qdrant provides superior performance, scalability, and features tailored for our needs.

1. Relational Database Schema (PostgreSQL with Prisma)

This schema translates the core business logic into a set of interconnected models. The structure is designed to be multi-tenant, secure, and fully auditable.

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

// =========================================
// Core Tenancy & User Management
// =========================================

/// Represents an enterprise client's isolated environment. All data is scoped to a Tenar
model Tenant {
  id    String @id @default(cuid())
  name  String @unique
  users User[]

  workflowDefinitions WorkflowDefinition[]
  workflowRuns        WorkflowRun[]
  credentials         Credential[]
  documents           Document[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

/// Represents a user within a Tenant.
model User {
  id       String  @id @default(cuid())
  email    String  @unique
  name     String?
  password String  // Will be a hashed password

  role Role @default(OPERATOR)

  tenantId String
  tenant   Tenant @relation(fields: [tenantId], references: [id])

  assignedHITLTasks HITLTask[]
  auditLogs         AuditLog[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

enum Role {
  ADMINISTRATOR /// Manages users, billing, and system settings
  DESIGNER      /// Creates and edits workflows
  OPERATOR      /// Runs workflows and handles HITL tasks
}


// =========================================
// Workflow Definition & Composition
// =========================================

/// Stores the design and structure of a workflow from the Workflow Studio.
model WorkflowDefinition {
  id          String   @id @default(cuid())
  name        String
  description String?
  status      WorkflowStatus @default(DRAFT)

  /// The JSON representation of the workflow graph (nodes, edges, agent configs).
  definitionGraph Json

  tenantId String
  tenant   Tenant @relation(fields: [tenantId], references: [id])

  runs WorkflowRun[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

enum WorkflowStatus {
  DRAFT
  ACTIVE
  ARCHIVED
}


// =========================================
// Workflow Execution & State
// =========================================

/// An individual, live instance of a running WorkflowDefinition.
model WorkflowRun {
  id String @id @default(cuid())

  workflowDefinitionId String
  workflowDefinition   WorkflowDefinition @relation(fields: [workflowDefinitionId], refe

  tenantId String
  tenant   Tenant @relation(fields: [tenantId], references: [id])

  status RunStatus @default(PENDING)

  /// The current data payload being passed through the workflow.
  payload Json?
  /// The internal state of the workflow engine for this run (e.g., current step).
  state   Json?

  startedAt   DateTime?
  completedAt DateTime?

  hitlTasks HITLTask[]
  auditLogs AuditLog[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

enum RunStatus {
  PENDING
  RUNNING
  COMPLETED
  FAILED
  PAUSED_FOR_APPROVAL
  PAUSED_WITH_ERROR
```

```
}


// ========================================
// Human-in-the-Loop (HITL) & Feedback
// ========================================

/// Represents a task that requires human review and action.
model HITLTask {
  id String @id @default(cuid())

  workflowRunId String
  workflowRun   WorkflowRun @relation(fields: [workflowRunId], references: [id])

  assignedToId String
  assignedTo   User @relation(fields: [assignedToId], references: [id])

  status      HITLStatus @default(PENDING)

  /// The data that needs to be reviewed by the human.
  taskData    Json

  /// The original agent output, logged for feedback analysis.
  originalAgentOutput Json?

  /// The corrected data from the human, if edited. This is critical for the learning loo
  humanCorrectedOutput Json?

  rejectionReason String?

  createdAt DateTime @default(now())
  resolvedAt DateTime?
}

enum HITLStatus {
  PENDING
  APPROVED
  REJECTED
  EDITED_AND_APPROVED
}


// ========================================
// Auditing & Logging
// ========================================

/// An immutable log of every significant action in the system.
model AuditLog {
  id String @id @default(cuid())

  workflowRunId String
  workflowRun   WorkflowRun @relation(fields: [workflowRunId], references: [id])

  /// Can be a User or null if the action was taken by the system/agent.
  actorId   String?
  actor     User?   @relation(fields: [actorId], references: [id])
  actorType String  @default("SYSTEM") // e.g., "USER", "AGENT", "SYSTEM"

  action    String // e.g., "WORKFLOW_STARTED", "STEP_APPROVED", "AGENT_ERROR"
  details   Json?  // Contextual details about the action

  timestamp DateTime @default(now())
}


// =========================================
// Integration & Document Management
// =========================================

/// Stores encrypted credentials for third-party systems.
model Credential {
  id           String @id @default(cuid())
  tenantId     String
  tenant       Tenant @relation(fields: [tenantId], references: [id])

  type         String // e.g., "OAUTH2", "API_KEY"
  serviceName  String // e.g., "SALESFORCE", "GMAIL"

  /// The credential value, encrypted at the application layer before being stored.
  encryptedValue String

  createdAt DateTime @default(now())
}

/// Represents a document indexed in the vector DB for RAG.
/// This model provides a relational link to the vectorized data.
model Document {
  id String @id @default(cuid())

  tenantId String
  tenant   Tenant @relation(fields: [tenantId], references: [id])

  sourceUri   String @unique // e.g., sharepoint://... or file://...
  fileName    String
  fileType    String // e.g., "PDF", "DOCX"

  status      IndexingStatus @default(PENDING)
  lastIndexed DateTime?

  createdAt DateTime @default(now())
}

enum IndexingStatus {
  PENDING
  INDEXING
  COMPLETED
  FAILED
}
```

2. Vector Database "Schema" (Qdrant)

A vector database doesn't have a rigid schema in the same way a relational database does. Instead, we define the structure of the data points we will store in a "collection." For our RAG and semantic search feature, each data point will represent a chunk of a document.

Here is the logical structure for a data point stored in our Qdrant collection:

```
{
  // Unique identifier for the vector point (e.g., a UUID).
  "id": "a1b2c3d4-e5f6-7890-1234-567890abcdef",

  // The high-dimensional vector embedding of the 'content' field.
  // This is what Qdrant uses for similarity search.
  "vector": [0.0123, -0.0456, ..., 0.0789],

  // The metadata payload. This is critical for filtering and security.
  "payload": {
    // The original text chunk from the document.
    "content": "The Q3 2025 sales target for the West region is $2.5 million, an increase

    // Link back to the relational database for more context.
    "document_id": "doc_2alkj34hkljha",

    // The unique ID of the tenant this data belongs to.
    // **This is a mandatory filter for ALL queries to ensure data isolation.**
    "tenant_id": "tnt_1a2b3c4d5e6f",

    // The source of the document chunk.
    "source_uri": "sharepoint://FinanceDocs/2025/Q3_Sales_Targets.docx",

    // **Security-critical field for permission-aware retrieval.**
    // An array of roles or groups that are allowed to see this data chunk.
    "allowed_roles": ["Finance", "SalesManager", "Executive"],

    // Other useful metadata for filtering.
    "author": "john.doe@enterprise.com",
    "created_at_source": "2025-07-15T10:00:00Z"
  }
}
```

This hybrid schema provides a robust, scalable, and secure foundation to build the full suite of features defined in the business logic for the Enterprise Agentic Workflow Engine.

⁂