```
adaptive_app
 ├── assessment.py
 ├── crud.py
 ├── database.py
 ├── main.py
 ├── models.py
 ├── requirements.txt
 ├── schemas.py
 ├── services.py
 └── settings.py
```

1. **assessment.py**

```
"""Asynchronous assessment micro-service (placeholder implementation).
  In production wire this to Azure OpenAI GPT-4o.
"""
from uuid import UUID
from datetime import datetime
from sqlalchemy.orm import Session

from models import Session as DbSession, Question
from schemas import SessionReport
from services import grade_answer, simple_report


def finalise_session(db: Session, session_id: UUID, answers: list[tuple[UUID, int]]):
    """answers = [(question_id, chosen_index), ...]"""
    correct = 0
    total = len(answers)

    for qid, idx in answers:
        q: Question = db.query(Question).get(qid)
        if grade_answer(q, idx):
            correct += 1

    pct = round(correct / total * 100, 1) if total else 0.0

    # close session row
    sess = db.query(DbSession).get(session_id)
    if sess:
        sess.ended_utc = datetime.utcnow()
        db.commit()

    return simple_report(score_pct=pct)
```

2. **crud.py**

```
"""Low-level DB helpers using SQLAlchemy ORM."""
from uuid import UUID, uuid4
from datetime import datetime
from sqlalchemy.orm import Session
from sqlalchemy.orm import joinedload
from models import (
    Topic, Subtopic, Question, Choice, Concept, Reference, User, Session as DbSession
)

# ---------------------------------------------------------------------------
# Study-plan retrieval
# ---------------------------------------------------------------------------

def fetch_study_plan(db: Session, topic_id: UUID):
    """Return Topic with eager-loaded sub-objects."""
    return (
        db.query(Topic)
        .filter(Topic.topic_id == topic_id)
        .options(
            # subtopics, concepts, refs, questions, choices all loaded in one go
        )
        .one_or_none()
    )

# ---------------------------------------------------------------------------
# Session management
# ---------------------------------------------------------------------------

def start_session(db: Session, user_id: UUID, topic_id: UUID) -> UUID:
    sess = DbSession(session_id= uuid4(),user_id=user_id, topic_id=topic_id)
    db.add(sess)
    db.commit()
    db.refresh(sess)
    return sess.session_id


def close_session(db: Session, session_id: UUID, ended_utc: datetime):
    sess = db.query(DbSession).get(session_id)
    if sess:
        sess.ended_utc = ended_utc
        db.commit()
```

3. database.py

```
"""
Central DB engine & session factory.
Uses the DATABASE_URL you've already defined in settings.py
"""
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base
from settings import DATABASE_URL

engine = create_engine(
    DATABASE_URL,
    pool_pre_ping=True,
    connect_args={
        "fast_executemany": True,
    },
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

4. main.py

```
from uuid import UUID
from fastapi import FastAPI, Depends, HTTPException, status
from sqlalchemy.orm import Session

from database import SessionLocal, Base, engine
from schemas import StudyPlanOut, AnswerIn, AnswerOut, StartSessionIn, SessionReport
from services import build_plan_payload, grade_answer
from models import Question
from crud import start_session
from assessment import finalise_session
# make sure metadata uses schema "cme"
Base.metadata.create_all(bind=engine)  # no-op if tables already exist

app = FastAPI(title="Adaptive Learning API", version="0.6.0")


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# ------------------------ ENDPOINTS --------------------------------------

@app.post("/api/session", response_model=UUID, summary="Start a study session")
def api_start_session(payload: StartSessionIn, db: Session = Depends(get_db)):
    sid = start_session(db, payload.user_id, payload.topic_id)
    return sid


@app.get("/api/lesson/{topic_id}/{user_id}", response_model=StudyPlanOut)
def api_get_plan(topic_id: UUID, user_id: UUID, db: Session = Depends(get_db)):
    plan = build_plan_payload(db, topic_id, user_id)
    if not plan:
        raise HTTPException(status_code=404, detail="Topic not found")
    return plan


@app.post("/api/answer", response_model=AnswerOut)
def api_answer(ans: AnswerIn, db: Session = Depends(get_db)):
    q: Question = db.query(Question).get(ans.question_id)
    if q is None:
        raise HTTPException(status_code=404, detail="Question not found")
    correct = grade_answer(q, ans.chosen_index)
    return AnswerOut(correct=correct)


@app.post("/api/report/{session_id}", response_model=SessionReport)
def api_report(session_id: UUID, db: Session = Depends(get_db)):
    # In a real build you'd pass the array of answers from frontend → backend
    # Here we fake it with an empty list just to show wiring.
    return finalise_session(db, session_id, answers=[])
```

5. **models.py**

```
"""
Minimal ORM layer matching the study-plan + progress schema.
Only the tables the adaptive-learning app needs are listed here.
If you later add exam-prep entities, extend this file and run Alembic.
"""
from sqlalchemy import (
    Column, String, Integer, Boolean, ForeignKey, DateTime, UniqueConstraint, TIMESTAMP,
text
)
from sqlalchemy.types import Text
from sqlalchemy.dialects.mssql import UNIQUEIDENTIFIER, TINYINT, NVARCHAR
from sqlalchemy.orm import relationship
from database import Base


class Topic(Base):
    __tablename__ = "topics"
    __table_args__ = {'schema': 'cme'}

    topic_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='topic_id')
    topic_name = Column(NVARCHAR(255), nullable=False, name='topic_name')
    created_utc = Column(DateTime, nullable=False, name='created_utc',
server_default=text("SYSUTCDATETIME()"))
    schema_version = Column(TINYINT, nullable=False, name='schema_version',
server_default=text("1"))

    # relationships
    subtopics = relationship("Subtopic", back_populates="topic", lazy="joined")
    study_plans = relationship("StudyPlan", back_populates="topic", lazy="select")


class Subtopic(Base):
    __tablename__ = "subtopics"
    __table_args__ = {'schema': 'cme'}

    subtopic_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='subtopic_id')
    topic_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.topics.topic_id"), nullable=False,
name='topic_id')
    title = Column(NVARCHAR(255), nullable=False, name='title')
    sequence_no = Column(Integer, nullable=True, name='sequence_no')
    status = Column(String(20), nullable=False, name='status')

    # relationships
    topic = relationship("Topic", back_populates="subtopics")
    concepts = relationship("Concept", back_populates="subtopic", lazy="select")
    questions = relationship("Question", back_populates="subtopic", lazy="select")
    subtopic_references = relationship("SubtopicReference", back_populates="subtopic",
lazy="select")


class Concept(Base):
    __tablename__ = "concepts"
    __table_args__ = {'schema': 'cme'}

    concept_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='concept_id')
    subtopic_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.subtopics.subtopic_id"),
nullable=False, name='subtopic_id')
    content = Column(Text, nullable=False, name='content')
    token_count = Column(Integer, nullable=True, name='token_count')

    # relationships
    subtopic = relationship("Subtopic", back_populates="concepts")


class Reference(Base):
    __tablename__ = "references"
    __table_args__ = {'schema': 'cme'}

    reference_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='reference_id')
    source_id = Column(NVARCHAR(128), nullable=False, name='source_id')
    citation_link = Column(NVARCHAR(512), nullable=True, name='citation_link')
    excerpt = Column(Text, nullable=False, name='excerpt')

    # relationships
    question_references = relationship("QuestionReference", back_populates="reference",
lazy="select")
    subtopic_references = relationship("SubtopicReference", back_populates="reference",
lazy="select")


class Question(Base):
    __tablename__ = "questions"
    __table_args__ = {'schema': 'cme'}

    question_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='question_id')
    subtopic_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.subtopics.subtopic_id"),
nullable=False, name='subtopic_id')
    stem = Column(Text, nullable=False, name='stem')
    explanation = Column(Text, nullable=False, name='explanation')
    correct_choice = Column(NVARCHAR(255), nullable=False, name='correct_choice',
server_default=text("''"))
```

```
    # relationships
    subtopic = relationship("Subtopic", back_populates="questions")
    choices = relationship("Choice", back_populates="question", lazy="joined")
    question_references = relationship("QuestionReference", back_populates="question",
lazy="select")
    variants = relationship("Variant", back_populates="question", lazy="select")


class Choice(Base):
    __tablename__ = "choices"
    __table_args__ = {'schema': 'cme'}

    question_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.questions.question_id"),
primary_key=True, name='question_id')
    choice_index = Column(TINYINT, primary_key=True, name='choice_index')
    choice_text = Column(NVARCHAR(255), nullable=False, name='choice_text')
    choice_id = Column(UNIQUEIDENTIFIER, nullable=False, name='choice_id',
server_default=text("NEWID()"))

    # relationships
    question = relationship("Question", back_populates="choices")


class Variant(Base):
    __tablename__ = "variants"
    __table_args__ = {'schema': 'cme'}

    variant_id = Column(UNIQUEIDENTIFIER, primary_key=True, name='variant_id')
    question_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.questions.question_id"),
nullable=False, name='question_id')
    variant_no = Column(TINYINT, nullable=False, name='variant_no')
    stem = Column(Text, nullable=False, name='stem')
    correct_choice_index = Column(TINYINT, nullable=False, name='correct_choice_index')

    # relationships
    question = relationship("Question", back_populates="variants")


class QuestionReference(Base):
    __tablename__ = "question_references"
    __table_args__ = {'schema': 'cme'}

    question_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.questions.question_id"),
primary_key=True, name='question_id')
    reference_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.references.reference_id"),
primary_key=True, name='reference_id')

    # relationships
    question = relationship("Question", back_populates="question_references")
    reference = relationship("Reference", back_populates="question_references")
```

```
class SubtopicReference(Base):
   __tablename__ = "subtopic_references"
   __table_args__ = {'schema': 'cme'}

   subtopic_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.subtopics.subtopic_id"),
primary_key=True, name='subtopic_id')
   reference_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.references.reference_id"),
primary_key=True, name='reference_id')

   # relationships
   subtopic = relationship("Subtopic", back_populates="subtopic_references")
   reference = relationship("Reference", back_populates="subtopic_references")


class StudyPlan(Base):
   __tablename__ = "study_plans"
   __table_args__ = {'schema': 'cme'}

   topic_id = Column(UNIQUEIDENTIFIER, ForeignKey("cme.topics.topic_id"),
primary_key=True, name='topic_id')
   assembled_utc = Column(DateTime, nullable=False, name='assembled_utc')
   plan_json = Column(Text, nullable=False, name='plan_json')

   # relationships
   topic = relationship("Topic", back_populates="study_plans")


class User(Base):
   __tablename__ = "users"
   __table_args__ = {"schema": "cme"}

   user_id     = Column(UNIQUEIDENTIFIER, primary_key=True)
   email       = Column(String(255), unique=True, nullable=False)
   display_name = Column(String(255))
   created_utc  = Column(DateTime, nullable=False,
server_default=text("SYSUTCDATETIME()"))
   sessions    = relationship("Session", back_populates="user")


class Session(Base):
   __tablename__ = "sessions"
   __table_args__ = {"schema": "cme"}

   session_id   = Column(UNIQUEIDENTIFIER, primary_key=True)
   user_id      = Column(UNIQUEIDENTIFIER, ForeignKey("cme.users.user_id"), nullable=False)
   topic_id     = Column(UNIQUEIDENTIFIER, ForeignKey("cme.topics.topic_id"),
nullable=False)
   started_utc  = Column(DateTime, nullable=False,
server_default=text("SYSUTCDATETIME()"))
```

```
  ended_utc    = Column(DateTime)
  user         = relationship("User", back_populates="sessions")
```

6. schemas.py

```
from typing import List, Optional
from uuid import UUID
from datetime import datetime
from pydantic import BaseModel, Field

# -------------------
# Study-plan payloads
# -------------------
class ChoiceOut(BaseModel):
  choice_index: int
  text: str

  class Config:
    orm_mode = True

class QuestionOut(BaseModel):
  question_id: UUID = Field(..., alias="question_id")
  stem: str
  explanation: str
  choices: List[ChoiceOut]

  class Config:
    orm_mode = True

class SubtopicOut(BaseModel):
  subtopic_id: UUID = Field(..., alias="subtopic_id")
  title: str
  concept: str
  references: List[str]
  questions: List[QuestionOut]

  class Config:
    orm_mode = True

class StudyPlanOut(BaseModel):
  topic_id: UUID
  topic_name: str
  percentage_complete: float
  subtopics: List[SubtopicOut]

# -------------------
# Session / answer I/O
# -------------------
class StartSessionIn(BaseModel):
  user_id: UUID
  topic_id: UUID
```

```
class AnswerIn(BaseModel):
    session_id: UUID
    subtopic_id: UUID
    question_id: UUID
    chosen_index: int

class AnswerOut(BaseModel):
    correct: bool

class SessionReport(BaseModel):
    session_id: UUID
    finished_utc: datetime
    score_pct: float
    strong_areas: List[str]
    focus_areas: List[str]
```

7. **services.py**

```
from uuid import UUID
from datetime import datetime
from sqlalchemy.orm import Session, joinedload

from crud import fetch_study_plan, start_session, close_session
from schemas import StudyPlanOut, SubtopicOut, QuestionOut, ChoiceOut, SessionReport
from models import Question, Choice, User

# ---------------------------------------------------------------------------
# Study-plan assembly
# ---------------------------------------------------------------------------

def build_plan_payload(db: Session, topic_id: UUID, user_id: UUID) -> StudyPlanOut:
    topic = fetch_study_plan(db, topic_id)
    if topic is None:
        return None

    # completion logic placeholder: 0.0 for MVP
    pct = 0.0

    return StudyPlanOut(
        topic_id=topic.topic_id,
        topic_name=topic.topic_name,
        percentage_complete=pct,
        subtopics=[
            SubtopicOut(
                subtopic_id=s.subtopic_id,
                title=s.title,
                concept=s.concepts[0].content if s.concepts else "",
                references=[r.reference.excerpt for r in s.subtopic_references],
                questions=[
                    QuestionOut(
                        question_id=q.question_id,
                        stem=q.stem,
                        explanation=q.explanation,
                        choices=[
                            ChoiceOut(choice_index=c.choice_index, text=c.choice_text)
                            for c in q.choices
                        ],
                    )
                    for q in s.questions
                ],
            )
            for s in topic.subtopics
        ],
    )

# ---------------------------------------------------------------------------
# Answer grading & simple rule-based assessment
# ---------------------------------------------------------------------------
# -------------------------------------------------
#  keep your new DB-aware version
# -------------------------------------------------
def grade_answer_db(db: Session, question_id: UUID, chosen_index: int) -> bool:
    """Compare chosen index to correct_choice string stored on the question."""
    q: Question = (
        db.query(Question)
        .filter(Question.question_id == question_id)
        .options(joinedload(Question.choices))
        .one()
    )

    try:
        chosen_text = next(
            c.choice_text for c in q.choices if c.choice_index == chosen_index
        )
    except StopIteration:
        raise ValueError("Invalid choice_index")

    return chosen_text.strip() == q.correct_choice.strip()


# -------------------------------------------------
#  thin adapter so main.py does not break
# -------------------------------------------------
def grade_answer(q: Question, chosen_index: int) -> bool:
    """Adapter kept for backward-compatibility with main.py."""
    try:
        chosen_text = next(
            c.choice_text for c in q.choices if c.choice_index == chosen_index
        )
    except StopIteration:
        raise ValueError("Invalid choice_index")
    return chosen_text.strip() == q.correct_choice.strip()
```

```
def simple_report(score_pct: float) -> SessionReport:
    # Placeholder – plug GPT-based assessment later
    strong = ["Core concepts"] if score_pct >= 80 else []
    focus = ["Review references"] if score_pct < 80 else []
    return SessionReport(
        finished_utc=datetime.utcnow(),
        score_pct=score_pct,
        strong_areas=strong,
        focus_areas=focus,
    )
```

8. **settings.py**

```
"""Central place to load env vars & direct configuration."""
import os
from functools import lru_cache
from typing import Optional

# Set this to False to skip Key Vault entirely and use direct values
USE_KEY_VAULT = False

# If you want to use Key Vault, set this to True and ensure proper authentication
KV_URL = os.getenv("KEYVAULT_URL") or "https://cme0207.vault.azure.net/"

DATABASE_URL =
"mssql+pyodbc://localhost\\MSSQLSERVER03/CME2?driver=ODBC+Driver+18+for+SQL+Serv
er&Trusted_Connection=yes&Encrypt=no&TrustServerCertificate=yes"
@lru_cache
def get_secret(name: str) -> Optional[str]:
    # First, try environment variables
    env_value = os.getenv(name)
    if env_value:
        return env_value

    # Skip Key Vault if disabled
    if not USE_KEY_VAULT:
        return None

    # Only try Key Vault if enabled and we have a URL
    if not KV_URL:
        return None

    try:
        from azure.identity import DefaultAzureCredential
        from azure.keyvault.secrets import SecretClient

        credential = DefaultAzureCredential()
        client = SecretClient(vault_url=KV_URL, credential=credential)
        # Convert underscore to hyphen for Key Vault naming convention
        kv_name = name.replace("_", "-")
        return client.get_secret(kv_name).value
    except Exception as e:
        # Log the error if needed, but don't crash
        print(f"Warning: Could not retrieve secret '{name}' from Key Vault: {e}")
        return None


class Settings:
    # SQLAlchemy URL format for SQL Server (converted from ODBC connection string)
    sql_connection: str = (
        get_secret("SQL_CONNECTION_STRING") or

"mssql+aioodbc://localhost\\MSSQLSERVER03/CME?driver=ODBC+Driver+18+for+SQL+Serv
er&Trusted_Connection=yes&Encrypt=no&TrustServerCertificate=yes"
    )

    openai_api_key: str = (
        get_secret("AZURE_OPENAI_KEY") or

"CzrrWvXbsmYcNguU1SqBpE9HDhhbfYsbkq3UedythCYCV9zNQ4mLJQQJ99BEACHYHv6XJ3w3
AAABACOGiIPm"
    )

    openai_endpoint: str = (
        get_secret("AZURE_OPENAI_ENDPOINT") or
        "https://azure1405.openai.azure.com/"
    )

    openai_deployment: str = "gpt-4o-2025-06-01"  # deployment name


settings = Settings()
```