# PART 2: Backend Engineering

## Backend Architecture

## Technologies

- Framework: FastAPI for high-performance asynchronous API
- Authentication: OAuth2 with JWT
- Validation: Pydantic for data validation
- API Documentation: Automatic Swagger/OpenAPI documentation
- Background Tasks: Celery or FastAPI background tasks

## API Design

1. RESTful Endpoints

```
/api/v1/conversations                # Collection endpoints
/api/v1/conversations/{id}           # Resource endpoints
/api/v1/conversations/{id}/messages  # Sub-resource endpoints
```

2. GraphQL API (Optional alternative)

```
type Conversation {
  id: ID!
  title: String
  messages: [Message!]!
  createdAt: DateTime!
}

type Query {
  conversation(id: ID!): Conversation
  conversations: [Conversation!]!
      }
```

## Code Organization

```
backend/
├── app/
│   ├── api/
│   │   ├── endpoints/
│   │   │   ├── auth.py
```

```
│   │   │   ├── conversations.py
│   │   │   ├── llm.py
│   │   │   └── users.py
│   │   ├── dependencies.py
│   │   └── router.py
│   ├── core/
│   │   ├── config.py
│   │   ├── security.py
│   │   └── logging.py
│   ├── db/
│   │   ├── models.py
│   │   └── repositories.py
│   ├── services/
│   │   ├── llm_service.py
│   │   ├── embedding_service.py
│   │   └── user_service.py
│   ├── schemas/
│   │   ├── conversation.py
│   │   ├── message.py
│   │   └── user.py
│   └── main.py
├── tests/
│   ├── api/
│   ├── services/
│   └── conftest.py
├── alembic/
│   └── versions/
├── Dockerfile
└── requirements.txt
```

## Best Practices

1. API Design:
    - Follow RESTful principles for resource management
    - Use versioned endpoints (e.g., `/api/v1/resource`)
    - Implement comprehensive error handling with meaningful status codes
    - Structure responses consistently with standardized formats
2. Dependency Injection:

```python
# dependencies.py
from fastapi import Depends
from sqlalchemy.orm import Session

from app.db.session import get_db
from app.services.llm_service import LLMService

def get_llm_service(db: Session = Depends(get_db)) -> LLMService:
    return LLMService(db)
```

- Use FastAPI's dependency injection system
- Create reusable dependencies for common functionality
- Implement scoped dependencies for request-level resources

3. Environment Configuration:

```python
# config.py
from pydantic import BaseSettings, PostgresDsn, SecretStr

class Settings(BaseSettings):
    API_V1_STR: str = "/api/v1"
    PROJECT_NAME: str = "LLM Application"
    POSTGRES_SERVER: str
    POSTGRES_USER: str
    POSTGRES_PASSWORD: SecretStr
    POSTGRES_DB: str
    SQLALCHEMY_DATABASE_URI: PostgresDsn = None

    # LLM Config
    LLM_API_KEY: SecretStr
    LLM_MODEL_NAME: str = "gpt-4"
    LLM_MAX_TOKENS: int = 2048

    class Config:
        env_file = ".env"

settings = Settings()
```

- Use environment variables for configuration
- Never hardcode sensitive information
- Implement configuration validation with Pydantic

4. Logging and Monitoring:

```python
# logging.py
import logging
import json
from datetime import datetime

class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_record = {
            "timestamp": datetime.utcnow().isoformat(),
            "level": record.levelname,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        }
        if hasattr(record, "correlation_id"):
            log_record["correlation_id"] = record.correlation_id
        return json.dumps(log_record)
```

```
def setup_logging():
    handler = logging.StreamHandler()
    handler.setFormatter(JSONFormatter())
    logging.basicConfig(
        handlers=[handler],
        level=logging.INFO
        )
```

- Implement structured logging
- Use correlation IDs to track requests across services
- Log appropriate information for debugging and auditing

5. Error Handling:

```
# errors.py
from fastapi import HTTPException, Request
from fastapi.responses import JSONResponse

class LLMServiceError(Exception):
    def __init__(self, message: str, code: str = "llm_error"):
        self.message = message
        self.code = code
        super().__init__(self.message)

async def llm_exception_handler(request: Request, exc: LLMServiceError):
    return JSONResponse(
        status_code=500,
        content={
            "error": {
                "code": exc.code,
                "message": exc.message,
                "request_id": request.state.request_id
            }
        }
    )

# In main.py
app.add_exception_handler(LLMServiceError, llm_exception_handler)
```

- Create custom exception classes
- Implement global exception handlers
- Return consistent error response formats
- Include request IDs in error responses

# Backend Infrastructure

## AWS Infrastructure

1. Compute Options
    - ECS Fargate for containerized services
    - EC2 instances with Auto Scaling Groups
    - Elastic Beanstalk for simplified deployment
    - Lambda for serverless microservices
2. Networking
    - VPC with public and private subnets
    - Application Load Balancer for traffic distribution
    - API Gateway for API management
    - WAF for security and rate limiting

## GCP Infrastructure

1. Compute Options
    - Cloud Run for containerized services
    - Compute Engine with managed instance groups
    - App Engine for simplified deployment
    - Cloud Functions for serverless components
2. Networking
    - VPC network configuration
    - Cloud Load Balancing
    - API Gateway for API management
    - Cloud Armor for security

## Best Practices

Infrastructure as Code (IaC):

```
# Example Terraform configuration for FastAPI service on ECS
resource "aws_ecs_task_definition" "fastapi_task" {
  family                   = "fastapi-service"
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = 256
  memory                   = 512
  execution_role_arn       = aws_iam_role.ecs_execution_role.arn

  container_definitions = jsonencode([{
    name      = "fastapi-container"
    image     = "${aws_ecr_repository.fastapi_repo.repository_url}:latest"
```

```
    essential = true

    portMappings = [{
      containerPort = 8000
      hostPort      = 8000
      protocol      = "tcp"
    }]

    environment = [
      { name = "POSTGRES_SERVER", value = aws_rds_cluster.postgres.endpoint
},
      { name = "POSTGRES_DB", value = "app" },
      { name = "POSTGRES_USER", value = "app_user" }
    ]

    secrets = [
      { name = "POSTGRES_PASSWORD", valueFrom =
aws_secretsmanager_secret.db_password.arn },
      { name = "LLM_API_KEY", valueFrom =
aws_secretsmanager_secret.llm_api_key.arn }
    ]

    logConfiguration = {
      logDriver = "awslogs"
      options = {
        "awslogs-group"         = aws_cloudwatch_log_group.fastapi_logs.name
        "awslogs-region"        = var.aws_region
        "awslogs-stream-prefix" = "fastapi"
      }
    }
  }])
      }
```

- Define infrastructure as code with Terraform, AWS CDK, or Pulumi
- Version control infrastructure definitions
- Use modules for reusable components
- Implement variable parameterization