🎯 Detailed Work Plan - AI Model Marketplace

Let me create a comprehensive, step-by-step plan with clear milestones, time estimates, and learning outcomes.

📊 Project Overview

Total Duration: 16-18 weeks
Working Style: Step-by-step with detailed explanations
End Goal: Production-ready AI Model Marketplace

Phase 1: Authentication Service (Week 1-2)

Week 1: Core Authentication

Day 1: Project Setup & Database Foundation (4-5 hours)

Session 1: Shared Configuration (1.5 hours)

What we'll build:

```
shared/
├── common/
│   ├── __init__.py
│   ├── database.py      # Database connection logic
│   └── logger.py        # Logging configuration
├── schemas/
│   ├── __init__.py
│   └── base.py          # Base Pydantic schemas
└── utils/
    ├── __init__.py
    └── helpers.py       # Common utility functions
```

Tasks:

1. Create base database connection class (AsyncSession)
2. Setup logging with JSON formatter
3. Create base Pydantic schemas with common fields
4. Test database connection to PostgreSQL

Learning outcomes:

- Async database connections
- Dependency injection pattern
- Structured logging
- Reusable code organization

Deliverable: Can connect to PostgreSQL from Python ✅

Session 2: Auth Service Structure (1 hour)

What we'll build:

```
services/auth/
├── __init__.py
├── main.py             # FastAPI app
└── config/
    ├── __init__.py
    └── settings.py      # Environment variables
```

Tasks:

1. Create FastAPI app instance
2. Setup environment variables with Pydantic Settings
3. Configure CORS
4. Add health check endpoint
5. Test server starts successfully

Code we'll write:

```
python
# main.py
from fastapi import FastAPI

app = FastAPI(title="Auth Service")

@app.get("/health")
```

```
def health_check():
    return {"status": "healthy"}
```

Learning outcomes:
- FastAPI application structure
- Environment variable management
- CORS configuration

Deliverable: Auth service runs on http://localhost:8001 ✅

Session 3: Database Models (1.5 hours)
What we'll build:
```
services/auth/models/
├── __init__.py
├── base.py          # Base model with common fields
└── user.py          # User model
```

Tasks:
1. Create Base model with UUID, timestamps, soft delete
2. Create User model with:
   - id (UUID)
   - email (unique)
   - password_hash
   - role (enum: publisher, consumer, admin)
   - is_verified (boolean)
   - created_at, updated_at, deleted_at
3. Add indexes for performance

Learning outcomes:
- SQLAlchemy declarative models
- UUID as primary keys
- Soft delete pattern
- Database indexes

Deliverable: User model defined ✅

Session 4: Database Migrations (1 hour)

What we'll build:
```
services/auth/
└── migrations/          # Alembic migrations
    ├── env.py
    ├── script.py.mako
    └── versions/
        └── 001_create_users_table.py
```

Tasks:

1. Initialize Alembic
2. Configure Alembic for async SQLAlchemy
3. Create first migration (users table)
4. Apply migration to database
5. Verify table exists in PostgreSQL

Commands:

```
bash
alembic init migrations
alembic revision --autogenerate -m "Create users table"
alembic upgrade head
```

Learning outcomes:
- Database migrations concept
- Alembic workflow
- Schema versioning

Deliverable: Users table exists in PostgreSQL ✅

Day 2: Authentication Logic (4-5 hours)

Session 1: Pydantic Schemas (1 hour)
What we'll build:
```
services/auth/schemas/
├── __init__.py
├── user.py              # User schemas
└── token.py             # Token schemas
```

Schemas to create:

1. `UserCreate` - Registration request
2. `UserLogin` - Login request
3. `UserResponse` - API response (no password!)
4. `Token` - JWT token response
5. `TokenPayload` - Token data

Example:

python
class UserCreate(BaseModel):
    email: EmailStr
    password: str = Field(min_length=8)

class UserResponse(BaseModel):
    id: UUID
    email: str
    role: str
    is_verified: bool
    created_at: datetime
```

**Learning outcomes:**
- Request validation
- Response serialization
- Pydantic Field validators
- Email validation

**Deliverable:** All schemas defined with validation ✅

---

#### Session 2: Security Module (1.5 hours)
**What we'll build:**
```
services/auth/services/
├── __init__.py
└── security.py   # Password & JWT utilities
```

**Functions to implement:**
1. `hash_password()` - Bcrypt hashing
2. `verify_password()` - Check password
```

3. `create_access_token()` - Generate JWT
4. `create_refresh_token()` - Long-lived token
5. `decode_token()` - Verify and decode JWT

Learning outcomes:
- Password hashing (never store plain text!)
- JWT structure (header, payload, signature)
- Token expiration
- Security best practices

Deliverable: Can hash/verify passwords, create/decode tokens ✅

Session 3: Auth Service Logic (1.5 hours)
What we'll build:
```
services/auth/services/
├── __init__.py
├── security.py
└── auth_service.py      # Business logic
```

Functions to implement:
1. `register_user()` - Create new user
2. `authenticate_user()` - Verify credentials
3. `get_user_by_email()` - Fetch user
4. `get_user_by_id()` - Fetch by ID
5. `verify_user_email()` - Mark as verified

Example flow:
```
User Registration:
1. Validate email format
2. Check if email already exists
3. Hash password
4. Create user in database
5. Return user (without password)
```

Learning outcomes:
- Business logic separation
- Database queries with SQLAlchemy
- Error handling

- Transaction management

Deliverable: Complete auth service logic ✅

Session 4: API Endpoints (1 hour)
What we'll build:
```
services/auth/api/
├── __init__.py
├── routes.py           # API endpoints
└── dependencies.py     # Reusable dependencies
```

Endpoints to create:

1. `POST /auth/register` - User registration
2. `POST /auth/login` - User login
3. `POST /auth/refresh` - Refresh token
4. `GET /auth/me` - Get current user (protected)
5. `POST /auth/logout` - Logout (invalidate token)

Example:

```
python
@router.post("/register", response_model=UserResponse)
async def register(
    user_data: UserCreate,
    db: AsyncSession = Depends(get_db)
):
    return await auth_service.register_user(db, user_data)
```

Learning outcomes:
- FastAPI routing
- Dependency injection
- Response models
- Status codes

Deliverable: All auth endpoints working ✅

Day 3: API Keys & Protection (3-4 hours)

Session 1: API Key Model (1 hour)
What we'll build:
```
services/auth/models/
├── __init__.py
├── user.py
└── api_key.py          # API Key model
```

API Key model fields:
- id (UUID)
- user_id (foreign key)
- key_hash (hashed API key)
- name (user-friendly name)
- scopes (JSON - permissions)
- last_used_at
- expires_at
- is_active
- created_at

Tasks:
1. Create APIKey model
2. Create migration
3. Apply migration

Learning outcomes:
- Foreign key relationships
- JSON fields in PostgreSQL
- API key security (hash, don't store plain)

Deliverable: API keys table in database ✅

Session 2: API Key Service (1 hour)
What we'll build:
```
services/auth/services/
├── __init__.py
├── security.py
├── auth_service.py
└── api_key_service.py   # API key logic
```

```

**Functions to implement:**
1. `generate_api_key()` - Create new key
2. `validate_api_key()` - Verify key
3. `list_user_keys()` - Get all keys for user
4. `revoke_api_key()` - Deactivate key
5. `rotate_api_key()` - Replace old key

**API Key format:**
```
aimpk_1234567890abcdef1234567890abcdef
^   ^
| └─ Random 32 characters
└─ Prefix (ai-marketplace-key)
```

**Learning outcomes:**
- API key generation
- Secure key storage
- Key rotation strategy

**Deliverable:** Can create and validate API keys ✅

---

#### Session 3: Protected Endpoints (1 hour)
**What we'll build:**
```
services/auth/api/
├── __init__.py
├── routes.py
└── dependencies.py   # Auth dependencies
```

Dependencies to create:

1. `get_current_user()` - From JWT token
2. `get_current_user_from_api_key()` - From API key
3. `require_role()` - Role-based access
4. `get_current_active_user()` - Verified users only

Usage example:

python
```

```
@router.get("/protected")
async def protected_route(
    current_user: User = Depends(get_current_user)
):
    return {"message": f"Hello {current_user.email}"}
```

Learning outcomes:
- FastAPI dependencies
- Authentication middleware
- Authorization patterns
- Role-based access control (RBAC)

Deliverable: Protected endpoints require auth ✅

Session 4: API Key Endpoints (1 hour)
Endpoints to create:
1. `POST /auth/api-keys` - Generate new key
2. `GET /auth/api-keys` - List user's keys
3. `DELETE /auth/api-keys/{key_id}` - Revoke key
4. `POST /auth/api-keys/{key_id}/rotate` - Rotate key

Learning outcomes:
- CRUD operations
- Path parameters
- Protected routes

Deliverable: Complete API key management ✅

Day 4: Testing & Error Handling (3-4 hours)

Session 1: Error Handling (1 hour)
What we'll build:
```
shared/common/
├── __init__.py
├── database.py
├── logger.py
└── exceptions.py        # Custom exceptions
```

Custom exceptions:

1. `UserAlreadyExistsError`
2. `InvalidCredentialsError`
3. `UserNotFoundError`
4. `InvalidTokenError`
5. `InactiveUserError`

Exception handler:

python
@app.exception_handler(UserAlreadyExistsError)
async def user_exists_handler(request, exc):
    return JSONResponse(
        status_code=400,
        content={"detail": "User already exists"}
)
```

**Learning outcomes:**
- Custom exceptions
- Global exception handlers
- Proper HTTP status codes
- Error response format

**Deliverable:** Proper error messages for all cases ✅

---

#### Session 2: Unit Tests (1.5 hours)
**What we'll build:**
```
tests/
├── __init__.py
├── conftest.py   # Pytest fixtures
└── unit/
    └── auth/
        ├── test_security.py
        ├── test_auth_service.py
        └── test_schemas.py

Tests to write:

1. Password hashing/verification

2. JWT creation/validation
3. User registration logic
4. Email validation
5. Token expiration

Example:

python
def test_password_hashing():
    password = "mysecurepassword"
    hashed = hash_password(password)
    assert verify_password(password, hashed)
    assert not verify_password("wrong", hashed)
```

**Learning outcomes:**
- Pytest basics
- Testing pure functions
- Assertions
- Test organization

**Deliverable:** Unit tests passing ✅

---

#### Session 3: Integration Tests (1.5 hours)
**What we'll build:**
```
tests/
└── integration/
    └── auth/
        ├── test_auth_api.py
        └── test_api_keys.py

Tests to write:

1. User registration flow
2. Login and get token
3. Access protected route
4. Refresh token flow
5. Create and use API key

Example:

python

```
async def test_user_registration(client):
    response = await client.post(
        "/auth/register",
        json={"email": "test@test.com", "password": "secure123"}
    )
    assert response.status_code == 201
    assert "id" in response.json()
```

Learning outcomes:
- API testing
- Test client setup
- Database fixtures
- Async testing

Deliverable: Integration tests passing ✅

Week 1 Deliverables Summary

✅ **Working Authentication Service**
- User registration
- Login with JWT
- Token refresh
- API key generation
- Protected endpoints
- Role-based access

✅ **Test Coverage**
- 80%+ code coverage
- Unit tests for logic
- Integration tests for API

✅ **Documentation**
- API docs (auto-generated by FastAPI)
- README for auth service
- Environment setup guide

Week 2: API Gateway (Advanced Patterns)

Day 5: Gateway Foundation (4-5 hours)

Session 1: Gateway Structure (1.5 hours)
What we'll build:
```
services/gateway/
├── __init__.py
├── main.py              # Gateway app
├── config/
│   └── settings.py
└── middleware/
    ├── __init__.py
    ├── auth.py          # Auth middleware
    ├── rate_limit.py    # Rate limiting
    └── logging.py       # Request logging
```

Tasks:
1. Create FastAPI gateway app
2. Setup proxy routing to auth service
3. Add request ID generation
4. Configure CORS for frontend

Learning outcomes:
- Reverse proxy pattern
- Request forwarding
- Middleware chains

Deliverable: Gateway forwards to auth service ✅

Session 2: Authentication Middleware (1.5 hours)
What we'll build:
- JWT validation middleware
- API key validation middleware
- User context injection

Flow:
```
Request → Gateway → Validate Token → Add User to Context → Forward to Service
```

Learning outcomes:
- Middleware pattern

- Context propagation
- Header manipulation

Deliverable: Gateway validates auth before forwarding ✅

Session 3: Rate Limiting (1 hour)
What we'll build:
```
services/gateway/middleware/
└── rate_limit.py
```

Implementation:
- Token bucket algorithm
- Redis for counter storage
- Per-user limits
- Per-endpoint limits

Learning outcomes:
- Rate limiting strategies
- Redis for counters
- DDoS protection

Deliverable: Rate limiting working ✅

Session 4: Request Logging (1 hour)
What we'll build:
- Log all requests
- Track latency
- Store in Elasticsearch
- Request/response IDs

Learning outcomes:
- Structured logging
- Performance monitoring
- Log aggregation

Deliverable: All requests logged ✅

Day 6: Service Discovery & Load Balancing (3-4 hours)

Session 1: Service Registry (1.5 hours)
What we'll build:
```
shared/common/
└── service_registry.py
```

Features:
- Register services on startup
- Health check pinging
- Service discovery
- Load balancing

Learning outcomes:
- Service discovery patterns
- Health checks
- Load balancing algorithms

Deliverable: Gateway knows all service locations ✅

Session 2: Circuit Breaker (1.5 hours)
What we'll build:
```
shared/common/
└── circuit_breaker.py
```

States:
- Closed (normal)
- Open (failing)
- Half-open (testing)

Learning outcomes:
- Circuit breaker pattern
- Failure handling
- Graceful degradation

Deliverable: Gateway handles service failures ✅

Day 7: Gateway Testing (3 hours)

Session 1: Gateway Tests (2 hours)
Tests:
- Request forwarding
- Auth validation
- Rate limiting
- Circuit breaker

Deliverable: Gateway tests passing ✅

Session 2: Load Testing (1 hour)
Using Locust:
- Simulate 100 concurrent users
- Test rate limits
- Measure latency
- Find bottlenecks

Deliverable: Performance baseline established ✅

Week 2 Deliverables Summary

✅ **Working API Gateway**
- Routes to all services
- Authentication enforcement
- Rate limiting
- Request logging
- Circuit breaker

✅ **Monitoring**
- Request metrics
- Service health
- Error tracking

Phase 2: Storage & Model Registry (Week 3-4)

Week 3: Storage Service & MinIO

Day 8: Storage Service (4 hours)

Session 1: MinIO Integration (2 hours)
What we'll build:
```
services/storage/
├── main.py
└── services/
    ├── minio_service.py
    └── file_service.py
```

Features:
- File upload (chunked)
- Presigned URLs
- File download
- File deletion

Learning outcomes:
- Object storage
- Chunked uploads
- Presigned URLs
- S3 compatibility

Deliverable: Can upload/download files ✅

Session 2: File Metadata (2 hours)
Database model:
- file_id
- user_id
- filename
- size
- content_type
- storage_path
- checksum (SHA256)

Learning outcomes:
- File metadata tracking
- Checksum validation
- Duplicate detection

Deliverable: File metadata stored ✅

Day 9: Model Registry Foundation (4 hours)

Session 1: MongoDB Setup (1 hour)
What we'll build:
```
services/model-registry/
└── config/
    └── mongodb.py
```

Tasks:

- MongoDB connection
- Collections creation
- Indexes setup

Deliverable: Connected to MongoDB ✅

Session 2: Model Schema (1.5 hours)

MongoDB document:

```
javascript
{
  model_id: UUID,
  owner_id: UUID,
  name: String,
  description: String,
  version: String,
  framework: Enum,
  tags: [String],
  pricing: {
    free_tier_limit: Number,
    price_per_request: Decimal
  },
  storage_path: String,
  status: Enum,
  created_at: Date
}
```

```

**Deliverable:** Model schema defined ✅

---

#### Session 3: Model Upload (1.5 hours)
**Endpoints:**
- `POST /models` - Upload model
- `GET /models/{id}` - Get model details
- `PUT /models/{id}` - Update model
- `DELETE /models/{id}` - Delete model (soft)

**Flow:**
```
1. Upload model file to Storage Service
2. Store metadata in MongoDB
3. Index in Elasticsearch
4. Return model details
```

Deliverable: Can upload models ✅

---

## Day 10: Model Versioning (3 hours)

Session 1: Version Management (2 hours)

Features:

- Semantic versioning (v1.2.3)
- Version history
- Default version
- Rollback capability

Deliverable: Model versioning working ✅

---

Session 2: Model Validation (1 hour)

Validation:

- File format check
- Size limits
```

- Framework detection
- Schema validation

Deliverable: Models validated on upload ✅

Week 4: Search & Discovery

Day 11: Elasticsearch Integration (4 hours)

Session 1: Indexing (2 hours)

Tasks:

- Define Elasticsearch mapping
- Index models on creation
- Update index on changes
- Remove from index on delete

Deliverable: Models indexed in Elasticsearch ✅

Session 2: Search API (2 hours)

Endpoints:

- Full-text search
- Filter by tags
- Filter by framework
- Sort by popularity
- Pagination

Deliverable: Search working ✅

Day 12-13: Marketplace Features (6 hours)

Features to implement:

- Model browsing
- Model detail page
- Model reviews
- Rating system

- Popular models
- Recently published
- Recommendations

Deliverable: Complete marketplace API ✅

Day 14: Testing Week 3-4 (4 hours)

Tests:

- File upload/download
- Model CRUD
- Search functionality
- Versioning

Deliverable: All tests passing ✅

Phase 3: Inference Service (Week 5-7)

Week 5: Basic Inference

Day 15-16: Model Loading (6 hours)

- ONNX Runtime integration
- Model caching
- GPU support
- Memory management

Deliverable: Can load and run ONNX models ✅

Day 17-18: Inference API (6 hours)

- Synchronous inference
- Batch inference
- Request queuing
- Response caching

Deliverable: Inference API working ✅

Week 6: Advanced Inference

Day 19-20: Async Inference (6 hours)

- Celery worker setup
- Job queue management
- Status tracking
- Webhook notifications

Deliverable: Long-running inference supported ✅

Day 21: Framework Support (4 hours)

- PyTorch models
- TensorFlow models
- Framework detection
- Automatic conversion

Deliverable: Multiple frameworks supported ✅

Week 7: Inference Optimization

Day 22-23: Performance (6 hours)

- Request batching
- Model warm-up
- Connection pooling
- Load testing

Deliverable: Optimized performance ✅

Day 24: Testing (4 hours)

- Inference tests

- Load tests
- Performance benchmarks

Deliverable: Inference thoroughly tested ✅

Phase 4: Billing & Payments (Week 8-9)

Week 8: Stripe Integration

Day 25-26: Payment Setup (6 hours)

- Stripe API integration
- Customer creation
- Payment methods
- Subscription plans

Deliverable: Can charge customers ✅

Day 27-28: Subscription Management (6 hours)

- Create subscriptions
- Cancel subscriptions
- Update plans
- Handle payment failures

Deliverable: Subscription system working ✅

Week 9: Usage-Based Billing

Day 29: Usage Tracking (4 hours)

- Track API calls
- Calculate costs
- Store in TimescaleDB
- Aggregations

Deliverable: Usage tracked ✅

Day 30-31: Invoicing (6 hours)

- Calculate monthly usage
- Generate invoices
- Charge customers
- Revenue distribution

Deliverable: Automatic billing working ✅

Day 32: Webhooks (4 hours)

- Handle Stripe webhooks
- Payment confirmation
- Subscription updates
- Failed payments

Deliverable: Webhooks processed ✅

Phase 5: Event-Driven Architecture (Week 10-11)

Week 10: Kafka Setup

Day 33-34: Event System (6 hours)

- Define event types
- Kafka producers
- Kafka consumers
- Event schemas

Deliverable: Events flowing through Kafka ✅

Day 35-36: Service Communication (6 hours)

- Services publish events
- Services consume events
- Event handlers
- Error handling

Deliverable: Services communicate via events ✅

Week 11: Saga Pattern

Day 37-39: Distributed Transactions (9 hours)

- Model purchase saga
- Payment → Access grant → Notification
- Compensating transactions
- Idempotency

Deliverable: Saga pattern implemented ✅

Day 40: Event Testing (4 hours)

- Event flow tests
- Saga tests
- Failure scenarios

Deliverable: Event system tested ✅

Phase 6: Notification & Analytics (Week 12)

Week 12: Supporting Services

Day 41-42: Notification Service (6 hours)

- Email sending
- Webhook delivery

- SMS (optional)
- Template management

Deliverable: Notifications working ✅

Day 43-44: Analytics Service (6 hours)

- Usage dashboards
- Revenue analytics
- Popular models
- User metrics

Deliverable: Analytics dashboards ✅

Phase 7: Monitoring & Observability (Week 13)

Week 13: Production Readiness

Day 45: Metrics (4 hours)

- Prometheus instrumentation
- Custom metrics
- Grafana dashboards
- Alerts

Deliverable: Metrics dashboard ✅

Day 46: Distributed Tracing (4 hours)

- OpenTelemetry setup
- Trace all services
- Jaeger UI
- Performance analysis

Deliverable: Full request tracing ✅

Day 47: Logging (3 hours)

- Structured logging
- Log aggregation
- Kibana dashboards
- Error tracking

Deliverable: Centralized logging ✅

Phase 8: Testing & Documentation (Week 14)

Week 14: Quality Assurance

Day 48-49: Comprehensive Testing (6 hours)

- Complete unit tests
- Integration tests
- E2E tests
- Load tests

Deliverable: 80%+ test coverage ✅

Day 50-51: Documentation (6 hours)

- API documentation
- Architecture diagrams
- Deployment guide
- User guides

Deliverable: Complete documentation ✅

Phase 9: Deployment (Week 15-16)

Week 15: Containerization

Day 52-53: Docker (6 hours)

- Dockerfiles for each service
- Multi-stage builds
- Image optimization
- Security scanning

Deliverable: All services containerized ✅

Day 54: Docker Compose (4 hours)

- Production compose file
- Environment configs
- Volume management
- Network setup

Deliverable: Full stack in Docker ✅

Week 16: Cloud Deployment

Day 55-57: Kubernetes (9 hours)

- Kubernetes manifests
- Helm charts
- Deployments
- Services & Ingress
- ConfigMaps & Secrets

Deliverable: K8s deployment ready ✅

Day 58-59: CI/CD (6 hours)

- GitHub Actions
- Automated testing
- Build pipelines
- Deployment automation

Deliverable: CI/CD pipeline working ✅

Day 60: Production Launch (4 hours)

- Deploy to cloud
- Smoke tests
- Monitoring setup
- Go live!

Deliverable: System running in production ✅

📊 Summary Statistics

Total Duration: 16 weeks (60 working days)
Estimated Hours: 240-300 hours
Services Built: 10 microservices
Technologies Mastered: 15+
Lines of Code: ~15,000-20,000

🎯 Key Milestones

- Week 2: First service (Auth) complete ✅
- Week 4: Can upload and search models ✅
- Week 7: Can run inference on models ✅
- Week 9: Payment system working ✅
- Week 11: Event-driven architecture ✅
- Week 14: Production-ready codebase ✅
- Week 16: Deployed to cloud ✅

📚 Skills You'll Learn

Backend Development

- FastAPI mastery
- Async Python
- RESTful API design
- Microservices architecture

Databases

- PostgreSQL (SQL)
- MongoDB (NoSQL)
- Redis (Cache)
- TimescaleDB (Time-series)
- Elasticsearch (Search)

Message Queues

- Kafka (Event streaming)
- RabbitMQ (Task queues)
- Celery (Background jobs)

Security

- JWT authentication
- API keys
- Password hashing
- RBAC

DevOps

- Docker
- Kubernetes
- CI/CD
- Monitoring (Prometheus, Grafana)
- Logging (ELK stack)
- Tracing (Jaeger)

Payments

- Stripe integration
- Subscription billing
- Usage-based billing
- Webhooks

Cloud & Storage

- MinIO/S3
- Object storage
- CDN concepts

Testing

- Unit testing
- Integration testing
- Load testing
- E2E testing

ML Integration

- Model serving
- ONNX Runtime
- PyTorch/TensorFlow
- GPU utilization