# CANDIDATE_INSTRUCTIONS

## AI Application Engineer - Take-Home Project

### Overview

Welcome to the technical assessment! This project evaluates your ability to build AI-powered applications using modern development tools and AI coding assistants. You'll create a sales forecasting system with backend ML pipeline, REST API, and a functional web interface.

**Time Allocation**: 2 days
**Tools Allowed**: Claude Code, Cursor, or any modern development tools
**Philosophy**: Focus on functionality and clear explanations

### Dataset

You've been provided with `ecommerce_sales_data.csv` containing 669 days of e-commerce sales data with the following columns:

- `date`: Daily timestamp (2023-01-01 to 2024-10-30)
- `daily_sales`: Daily sales revenue in USD
- `product_category`: Product category (Electronics)
- `marketing_spend`: Daily marketing budget in USD
- `day_of_week`: Day name (Monday-Sunday)
- `is_holiday`: Binary flag (1=holiday, 0=regular day)

### Your Task

Build a sales forecasting system with the following components:

### Part 1: Backend & ML Pipeline (100 points)

#### 1.1 Data Processing Pipeline (15 points)

Create a data processing system:

- Data validation and cleaning
- Feature engineering (at least: lag features, rolling averages, date features)
- Basic data quality checks

**Deliverables**:

- Data processing functions (can be in single file or module)
- Brief documentation of features created

### 1.2 Machine Learning Model(s) (35 points)

Implement at least ONE working forecasting model:

- **Choose one approach**: Prophet, SARIMA, XGBoost, LSTM, or similar
- Proper train/test split with temporal ordering (no shuffling!)
- Model evaluation with appropriate metrics (MAPE, MAE, RMSE)
- Save trained model for reuse

**Bonus** (+10 points):

- Compare 2 different model types
- Simple hyperparameter tuning

**Deliverables**:

- Model training script or notebook
- Saved model file(s)
- Model evaluation results showing performance

### 1.3 REST API (30 points)

Build a functional API using FastAPI or Flask:

**Core Endpoints** (Required):

```
POST /api/forecast
  - Input: {horizon: number of days}
  - Output: {predictions: [{date, value}]}

GET /api/historical
  - Output: historical sales data

GET /api/metrics
  - Output: model performance metrics (MAPE, MAE, RMSE)
```

**Nice to Have** (Optional):

```
GET /api/feature-importance
  - Output: feature importance if using tree-based model

POST /api/upload
  - Upload new CSV data
```

**Requirements**:

- Basic input validation
- Error handling with appropriate HTTP status codes
- CORS enabled for frontend
- API documentation (Swagger auto-generated is fine)

**Deliverables**:

- Working API with at least 3 core endpoints
- README with API usage examples

### 1.4 Data Storage (10 points)

Simple data persistence:

- **SQLite is fine** (no need for PostgreSQL)
- Store: historical_sales, predictions, model_metrics
- Basic schema (can be simple tables)

**Deliverables**:

- Database setup script or initial schema
- Functions to read/write data

### 1.5 Testing & Code Quality (10 points)

Basic quality assurance:

- At least a few unit tests for critical functions
- API endpoint tests (even just 2-3 key ones)
- Clean, readable code with comments
- Basic error handling

**Deliverables**:

- Test file(s) with some coverage
- Code that runs without errors

## Part 2: Frontend Web Application (60 points)

### 2.1 Functional Web Interface (40 points)

Build a simple web interface (React, Vue, Svelte, or even plain HTML+JS):

**Required Views** (Choose approach that works for you):

**Option A: Single-Page Dashboard** (Simpler):

- Display key metrics (total sales, average, model MAPE)
- Historical sales chart (line chart)
- Forecast chart showing predictions
- Simple form to generate new forecasts (input: number of days)
- Display historical data in table

**Option B: Multi-Page App** (More structure):

1. **Overview**: KPIs + historical chart + latest forecast
2. **Forecast**: Form to generate predictions + visualization
3. **Data View**: Table showing historical data + model metrics

**Deliverables**:

- Working web interface that connects to your API
- Charts display data correctly
- User can trigger forecast generation
- Basic responsiveness (doesn't break on mobile)

### 2.2 Data Visualization (15 points)

Functional charts (doesn't need to be fancy):

- Use any library: Chart.js, Plotly, Recharts, or even matplotlib-generated images
- Line chart for time series data
- Forecast should be visually distinct from historical data
- Tooltips showing values on hover (nice to have)
- Basic legend

**Focus on**:

- Charts render correctly
- Data is readable and understandable
- Clear labels and axes

### 2.3 Usability & Documentation (5 points)

Make it understandable:

- Clear labels and headings explaining what each section shows
- Loading indicator when fetching data
- Error messages if API calls fail

- Brief text explaining what the forecast means
- Instructions for using the interface

**Deliverables**:

- User can understand what they're looking at without guessing
- Basic error handling
- Simple, clean layout (CSS framework like Bootstrap/Tailwind is fine)

## Part 3: Documentation & Deployment (20 points)

### 3.1 Documentation (15 points)

Clear project documentation:

**README.md** must include:

- Project overview (what it does)
- Technology stack used
- **Setup instructions** (step-by-step to run locally)
- How to run the backend API
- How to run the frontend
- API endpoint examples
- Model performance results (MAPE, MAE, RMSE)
- **Your approach**: Brief explanation of your ML model choice and why

**Code Documentation**:

- Comments explaining complex logic
- Brief docstrings for main functions
- Explanation of key design decisions

**AI Usage Report** (1 page):

- How you used AI coding assistants
- Examples where you modified AI suggestions
- What worked well / what didn't

**Deliverables**:

- Complete README that someone can follow to run your project
- Code is understandable with comments

### 3.2 Deployment (5 points)

Make it easy to run:

**Minimum** (choose one):

- Simple `docker-compose up` that starts everything
- OR clear instructions to run backend + frontend separately
- Environment variables documented

**Nice to Have**:

- Requirements.txt / package.json with all dependencies
- Shell script to set up environment
- Live demo URL (bonus +5 points)

**Deliverables**:

- Can run your project by following README
- Dependencies clearly listed

## Bonus Features (Optional - 20+ extra points)

Only if you have time - these are nice to have:

### Easy Wins:

- Multiple model comparison (show 2 models side by side) (+5)
- Feature importance visualization (+3)
- Data upload endpoint to add new data (+4)
- Confidence intervals on forecasts (+4)
- Download forecast as CSV (+2)

### More Advanced:

- What-if analysis (adjust marketing spend, see impact) (+8)
- Anomaly detection highlighting (+6)
- Live demo deployed online (+5)
- Real-time model retraining trigger (+5)

## Technical Requirements

### Backend:

- **Language**: Python 3.8+
- **Framework**: FastAPI or Flask (your choice)
- **ML Libraries**: Choose what you're comfortable with

    - Prophet, SARIMA (statsmodels), XGBoost, scikit-learn, etc.
- **Database**: SQLite is fine (simple is good)
- **Testing**: pytest with a few key tests

### Frontend:

- **Framework**: React, Vue, Svelte, or plain HTML/JS - whatever you know
- **Styling**: Use any CSS framework (Bootstrap, Tailwind) or plain CSS
- **Charts**: Chart.js, Plotly, Recharts - pick one that's easy
- **HTTP Client**: Fetch API or Axios

### Code Quality:

- Clean, readable code with comments
- Basic error handling
- No hardcoded credentials
- Clear variable names
- Some tests for critical paths

## Suggested Project Structure

Keep it simple! Here's one way to organize:

```
sales-forecasting/
├── backend/
│   ├── api.py              # FastAPI/Flask app with endpoints
│   ├── data_processing.py  # Feature engineering
│   ├── model.py            # ML model training and prediction
│   ├── database.py         # SQLite setup and queries
│   ├── requirements.txt    # Python dependencies
│   └── tests/              # A few test files
│       └── test_api.py
├── frontend/
│   ├── index.html          # Main HTML file
│   ├── app.js              # JavaScript for API calls and UI
│   ├── style.css           # Styling
│   └── package.json        # If using npm packages
├── data/
│   └── ecommerce_sales_data.csv
├── models/                 # Saved model files
│   └── forecast_model.pkl
├── README.md
└── docker-compose.yml      # Optional: if using Docker
```

**Or organize however makes sense to you!** This is just a suggestion.

## Submission Guidelines

### What to Submit:

1. **Source Code**:
    - Complete backend and frontend codebases
    - All configuration files
    - Database migration scripts
    - Docker configuration
2. **Documentation**:
    - README.md with setup instructions
    - ARCHITECTURE.md with design decisions
    - API.md with endpoint documentation
    - Code comments for complex logic
3. **Tests**:
    - Backend unit and integration tests
    - Test coverage report
    - API testing collection (Postman/Insomnia)
4. **Deployment**:
    - Docker setup with instructions
    - OR live demo URL (highly recommended)
    - Environment configuration examples
5. **AI Usage Report** (1-2 pages):
    - How you used AI coding assistants
    - Examples of prompts you used
    - Instances where you modified/rejected AI suggestions
    - What you learned from using AI tools

### Submission Format:

**Option A: GitHub Repository** (Preferred)

- Create a private GitHub repository
- Add comprehensive README
- Share access with interviewer email

- Include git history (meaningful commits)

**Option B: Zip File**

- Compress entire project
- Include all files except node_modules, **pycache**, venv
- Email download link (Google Drive, Dropbox, etc.)

**Before Submission Checklist:**

- [ ] Code runs without errors locally
- [ ] All tests pass
- [ ] README has complete setup instructions
- [ ] Environment variables documented
- [ ] Docker setup works (or live demo accessible)
- [ ] API documentation is complete
- [ ] Frontend is responsive and polished
- [ ] No sensitive data or API keys in code
- [ ] Git history is clean (if using Git)
- [ ] AI usage documented

## Evaluation Criteria

You will be assessed on a 180-point scale (200+ with bonuses):

| Category | Points | Key Focus |
|---|---|---|
| **Backend & ML** | 100 | Working model, API, basic tests |
| **Frontend** | 60 | Functional interface, readable charts |
| **Documentation** | 20 | Clear README, explainable code |
| **Bonus Features** | 20+ | Extra polish, advanced features |

**What We're Looking For:**

**Priority 1 - Functionality** (Must Have):

- ML model that works and beats naive baseline (MAPE < 20%)
- API endpoints that return correct data
- Frontend that displays forecasts clearly
- Project runs by following your README

**Priority 2 - Quality** (Important):

- Clean, understandable code
- Basic error handling
- Some tests for main functions
- Good documentation of your approach

**Priority 3 - Polish** (Nice to Have):

- Beautiful UI
- Advanced features
- High test coverage
- Live deployment

## Tips for Success

### Time Management (2 days = ~16 hours):

- **Day 1 (8 hours)**:
  - Hour 1-2: Data exploration, feature engineering
  - Hour 3-5: Train and evaluate ML model
  - Hour 6-8: Build basic API with 3 core endpoints
- **Day 2 (8 hours)**:
  - Hour 1-3: Build frontend with basic chart
  - Hour 4-5: Connect frontend to API
  - Hour 6-7: Write tests, add error handling
  - Hour 8: Documentation, README, final testing

### Strategy:

1. **Start Simple**: Get one model working before trying multiple
2. **MVP First**: End-to-end basic version, then add features
3. **Use AI Smart**: Generate boilerplate, but validate everything
4. **Test As You Go**: Don't wait until the end
5. **Document While Coding**: Comments and README as you build
6. **Prioritize**: Working > Perfect. Functionality > Beauty.

### AI Tool Usage:

- Use Claude Code or Cursor to accelerate development
- Have AI generate boilerplate and scaffolding
- Validate all AI suggestions before accepting
- Use AI for code review and optimization

- Document interesting AI interactions
- Show critical thinking - don't blindly accept

**Common Pitfalls to Avoid:**

- **Overengineering**: Don't build a distributed system for 669 rows of data
- **Perfect is the enemy of good**: A working basic system beats a half-finished fancy one
- **Ignoring time**: Track your hours, don't spend all day tuning hyperparameters
- **No testing**: At least test your API endpoints manually
- **Poor README**: If we can't run it, we can't evaluate it
- **AI Blindness**: Don't copy-paste without understanding

## Getting Started

1. **Review requirements thoroughly**
2. **Set up your development environment**
3. **Create project structure**
4. **Start with backend API + one model**
5. **Add frontend when backend is stable**
6. **Test and iterate**
7. **Deploy early, deploy often**
8. **Document throughout**

## Resources

You may use:

- Official documentation for any libraries
- Stack Overflow and similar resources
- AI coding assistants (Claude Code, Cursor, GitHub Copilot)
- Online tutorials and guides

You may NOT:

- Copy complete solutions from others
- Use pre-built forecasting platforms
- Share assignment with others

## Submission Deadline

**Due**: [Specific date/time will be provided]
**Late submissions**: Accepted with point deductions (10% per day)

---

**We're excited to see what you build! This is your chance to showcase your full-stack AI engineering skills. Good luck!**

**Questions or need more time?** Email us ASAP - we're reasonable about extensions for valid reasons.