TalentScout Hiring Assistant - Complete Documentation

Table of Contents

1. Introduction
2. System Architecture
3. Installation & Setup
4. Core Components
5. Data Models
6. Agent System
7. Conversation Flow
8. Data Validation
9. Storage System
10. UI Components
11. API Integration
12. Configuration Guide
13. Extending the System
14. Troubleshooting
15. Best Practices
16. Security & Privacy
17. Performance Optimization
18. Testing
19. Deployment
20. FAQ

---

1. Introduction

1.1 Purpose

TalentScout Hiring Assistant is an AI-powered conversational agent designed to automate the initial stages of candidate screening. It replaces traditional form-based applications with natural, conversational interactions while maintaining professional standards.

1.2 Key Benefits

- **Efficiency**: Reduces HR workload by 70%

- **Consistency**: Standardized screening process for all candidates
- **User Experience**: Natural conversation vs. boring forms
- **Data Quality**: Real-time validation ensures accurate information
- **Scalability**: Handle unlimited concurrent candidates

1.3 Use Cases

- Initial candidate screening for tech positions
- Pre-interview technical assessment
- Automated resume collection
- Skills verification
- Candidate pipeline management

---

2. System Architecture

2.1 High-Level Architecture

```
┌─────────────────────────────────────────────────────────────┐
│                 User Interface (Streamlit)                  │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐       │
│  │ Chat Display │  │ Input Field  │  │   Sidebar    │       │
│  └──────────────┘  └──────────────┘  └──────────────┘       │
└─────────────────────────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────────────────────────┐
│               Application Logic Layer                       │
│  ┌───────────────────────────────────────────────────────┐  │
│  │        Conversation State Management                  │  │
│  │  • Session State • Field Tracking • Validation        │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────────────────────────┐
│                 Agent Framework                             │
│  ┌───────────────────────────────────────────────────────┐  │
│  │  Technical Question Generator Agent                   │  │
│  │  • OpenAI-Compatible Interface                        │  │
│  │  • Pydantic Output Validation                         │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
```

```
                     │
                     ▼
┌─────────────────────────────────────────────────────────┐
│          External Services                               │
│  ┌──────────────────┐      ┌──────────────────────┐     │
│  │ Gemini AI API    │      │ Local File System    │     │
│  │ (via OpenAI SDK) │      │ (JSON Storage)       │     │
│  └──────────────────┘      └──────────────────────┘     │
└─────────────────────────────────────────────────────────┘
```

2.2 Technology Stack

Layer	Technology	Purpose
Frontend	Streamlit	Web UI framework
Backend	Python 3.8+	Core application logic
AI Model	Gemini 2.0 Flash	Question generation
Agent Framework	Custom Agent/Runner	AI orchestration
Data Validation	Pydantic	Type safety & validation
Async Runtime	asyncio, nest_asyncio	Async operations
Environment	python-dotenv	Configuration management
Storage	JSON (File System)	Data persistence

2.3 Component Interaction Flow

User Input → Streamlit → Session State → Validation →
→ Data Extraction → Agent (if needed) → Gemini API →
→ Response Generation → UI Update → Storage

---

3. Installation & Setup

3.1 System Requirements

Minimum Requirements:

- Python 3.8 or higher
- 4GB RAM
- 500MB disk space
- Internet connection (for Gemini API)

Recommended:

- Python 3.10+
- 8GB RAM
- SSD storage
- Stable internet connection

3.2 Detailed Installation

Step 1: Environment Setup

```
# Create project directory
mkdir talentscout-hiring-assistant
cd talentscout-hiring-assistant

# Create virtual environment
python -m venv venv

# Activate virtual environment
# On macOS/Linux:
source venv/bin/activate
# On Windows:
venv\Scripts\activate
```

Step 2: Install Dependencies

```
# Install core packages
pip install streamlit==1.28.0
pip install nest-asyncio==1.5.8
pip install pydantic==2.5.0
pip install openai==1.3.0
pip install python-dotenv==1.0.0

# Install agent framework (adjust based on your framework)
pip install agents

# Or install all at once from requirements.txt
pip install -r requirements.txt
```

Step 3: Configuration

```
# Create .env file
touch .env  # On Windows: type nul > .env

# Add API key to .env
echo "GEMINI_API_KEY=your_actual_api_key_here" >> .env
```

Step 4: Create Directory Structure

```
# Create data storage directory
mkdir candidate_data

# Verify structure
ls -la
# Should show: app.py, .env, candidate_data/, venv/
```

Step 5: Verify Installation

```
# Test import
python -c "import streamlit; import nest_asyncio; import pydantic; print('All imports successful')"

# Run application
streamlit run app.py
```

3.3 Getting Gemini API Key

1. Visit Google AI Studio
2. Sign in with Google account
3. Click "Get API Key"
4. Create new API key or use existing
5. Copy key to `.env` file

3.4 Troubleshooting Installation

Issue: Module not found errors

```
# Solution: Ensure virtual environment is activated
source venv/bin/activate  # or venv\Scripts\activate on Windows
pip list  # Verify packages are installed
```

Issue: Permission denied on Linux/Mac

```
# Solution: Fix permissions
chmod +x venv/bin/activate
```

Issue: Port 8501 already in use

```
# Solution: Use different port
streamlit run app.py --server.port 8502
```

---

4. Core Components

4.1 Session State Management

Session state maintains conversation context across interactions:

```
# Key session state variables
st.session_state.messages          # Chat history
st.session_state.candidate_info     # Collected data
st.session_state.current_field      # Active field being collected
st.session_state.tech_questions     # Generated questions
st.session_state.conversation_active # Conversation status
st.session_state.processing         # Processing flag
```

Lifecycle:

1. **Initialization**: First page load creates empty state
2. **Update**: Each user interaction updates relevant state
3. **Persistence**: State persists during session
4. **Reset**: Manual reset or session timeout clears state

4.2 Message System

Messages follow a structured format:

```
message = {
    "role": "assistant" | "user",
    "content": "message text"
}
```

Message Flow:

1. User submits input via `st.chat_input()`
2. Message added to `st.session_state.messages`

3. Processing logic generates response
4. Response added to messages
5. Both displayed in chat interface

4.3 Field Collection System

Collection Order:

full_name → email → phone → years_experience →
→ desired_position → current_location → tech_stack

State Transitions:

```
field_order = [
    "full_name",
    "email",
    "phone",
    "years_experience",
    "desired_position",
    "current_location",
    "tech_stack"
]
```

After completing tech_stack collection, system transitions to `"completed"` state for technical questions.

---

5. Data Models

5.1 CandidateInfo Model

```
class CandidateInfo(BaseModel):
    full_name: Optional[str] = None
    email: Optional[str] = None
    phone: Optional[str] = None
    years_experience: Optional[str] = None
    desired_position: Optional[str] = None
    current_location: Optional[str] = None
    tech_stack: Optional[List[str]] = None
```

Field Specifications:

Field	Type	Validation	Example
full_name	str	1-3 words, alphabetic	"John Doe"
email	str	RFC 5322 format	"john@example.com"
phone	str	10-15 digits	"1234567890"
years_experience	str	Numeric	"5"
desired_position	str	Length > 1	"Software Engineer"
current_location	str	Length > 1	"New York, USA"
tech_stack	List[str]	Known tech keywords	["python", "django"]

5.2 TechnicalQuestions Model

```
class TechnicalQuestions(BaseModel):
   questions: List[str]
   tech_category: str
```

Usage:

- Output type for AI agent
- Ensures structured responses
- Validates question format

5.3 Stored Data Structure

```
{
  "timestamp": "20250106_143022",
  "candidate_info": {
    "full_name": "John Doe",
    "email": "john.doe@example.com",
    "phone": "1234567890",
    "years_experience": "5",
    "desired_position": "Senior Backend Developer",
    "current_location": "San Francisco, USA",
    "tech_stack": ["python", "django", "postgresql", "redis"]
  },
  "technical_questions": [
    "Explain Django's ORM and how it handles database transactions",
    "How would you optimize a slow PostgreSQL query?",
    "Describe your approach to implementing caching with Redis"
```

```
 ],
 "candidate_answers": [
   "Django's ORM provides an abstraction layer...",
   "I would start by analyzing the query execution plan...",
   "I typically use Redis for session storage and cache..."
 ],
 "questions_and_answers": [
   {
     "question_number": 1,
     "question": "Explain Django's ORM...",
     "answer": "Django's ORM provides..."
   }
 ],
 "status": "initial_screening_complete"
}
```

---

6. Agent System

6.1 Technical Question Generator Agent

Configuration:

```
tech_question_agent = Agent(
   name="Technical Question Generator",
   instructions="...",
   model=gemini_model,
   output_type=TechnicalQuestions,
)
```

Instructions Breakdown:

1. **Role**: Expert technical interviewer
2. **Input**: Candidate's tech stack
3. **Output**: 3-5 relevant questions per technology
4. **Guidelines**:
   - Assess practical knowledge
   - Mix difficulty levels
   - Technology-specific questions
   - Open-ended format
   - Problem-solving focused

6.2 Agent Execution Flow

```
# Async execution
tech_result = asyncio.run(Runner.run(
    tech_question_agent,
    f"Generate technical questions for: {tech_stack_msg}"
))

# Extract results
questions = tech_result.final_output.questions[:5]
```

Error Handling:

- Fallback to generic question on failure
- Graceful degradation ensures conversation continues
- All exceptions caught and logged

6.3 Model Configuration

```
GEMINI_BASE_URL = "https://generativelanguage.googleapis.com/v1beta/openai/"
GEMINI_MODEL = "gemini-2.0-flash"

gemini_client = AsyncOpenAI(
    base_url=GEMINI_BASE_URL,
    api_key=google_api_key
)

gemini_model = OpenAIChatCompletionsModel(
    model=GEMINI_MODEL,
    openai_client=gemini_client
)
```

Why Gemini 2.0 Flash:

- Fast response times (< 2 seconds)
- Cost-effective for high volume
- Strong technical knowledge
- OpenAI-compatible API

---

7. Conversation Flow

7.1 Complete Flow Diagram

```
START
  │
  ├─→ Display Greeting
  │
  ├─→ INFORMATION COLLECTION PHASE
  │     │
  │     ├─→ Ask for Full Name
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     ├─→ Ask for Email
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     ├─→ Ask for Phone
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     ├─→ Ask for Years of Experience
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     ├─→ Ask for Desired Position
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     ├─→ Ask for Current Location
  │     │     └─→ Validate → [Valid] → Save
  │     │                → [Invalid] → Clarify
  │     │
  │     └─→ Ask for Tech Stack
  │          └─→ Validate → [Valid] → Save
  │                     → [Invalid] → Clarify
  │
  ├─→ TECHNICAL QUESTIONS PHASE
  │     │
  │     ├─→ Generate Questions (via AI Agent)
  │     │
  │     ├─→ Ask Question 1
  │     │     └─→ Capture Answer
  │     │
  │     ├─→ Ask Question 2
  │     │     └─→ Capture Answer
  |     |
  |     └─→ ... (repeat for all questions)
  |
  ├─→ COMPLETION PHASE
  |     |
  |     ├─→ Save All Data to JSON
  |     |
  |     ├─→ Display Thank You Message
  |     |
  |     └─→ End Conversation
  |
END
```

7.2 State Transitions

States:
- full_name
- email
- phone
- years_experience
- desired_position
- current_location
- tech_stack
- completed (technical questions)

7.3 Exit Handling

```
exit_keywords = [
    'bye', 'exit', 'quit', 'goodbye',
    'no thanks', 'end conversation', 'stop'
]

# Case-insensitive matching
if any(keyword in message.lower() for keyword in exit_keywords):
    # Display farewell
    # Set conversation_active = False
    # Stop further processing
```

---

8. Data Validation

8.1 Validation Functions

Full Name Validation

```
# Rules:
# - 1-3 words
# - All alphabetic characters
# - No numbers or special characters

words = content.split()
if 1 < len(words) <= 3 and all(w.isalpha() for w in words):
    extracted["full_name"] = content
```

Examples:

- ✅ "John Doe"
- ✅ "Mary Jane Watson"
- ❌ "John"
- ❌ "John123"
- ❌ "John Doe Smith Jr."

Email Validation

```
# Rules:
# - RFC 5322 compliant
# - Format: localpart@domain.tld
# - 2+ character TLD

pattern = r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"
if re.fullmatch(pattern, content):
    extracted["email"] = content
```

Examples:

- ✅ "john.doe@example.com"
- ✅ "user+tag@company.co.uk"
- ❌ "invalid@email"
- ❌ "no-at-sign.com"
- ❌ "@example.com"

Phone Validation

```
# Rules:
# - 10-15 digits only
# - No country code prefix symbols
# - No spaces or dashes
```

```
pattern = r"\d{10,15}"
if re.fullmatch(pattern, content):
    extracted["phone"] = content
```

Examples:

- ✅ "1234567890"
- ✅ "919876543210"
- ❌ "+1234567890"
- ❌ "123-456-7890"
- ❌ "12345" (too short)

Tech Stack Validation

```
# Rules:
# - Match against 200+ tech keywords
# - Case-insensitive matching
# - Extract all mentioned technologies

tech_keywords = ["python", "java", "javascript", ...]
matched_techs = [
    tech for tech in tech_keywords
    if tech in lower_content
]
```

Examples:

- ✅ "I know Python, Django, and PostgreSQL" → Extracts: ["python", "django", "postgresql"]
- ✅ "React.js and Node.js" → Extracts: ["react", "nextjs"] (if "nextjs" mentioned)

8.2 Technology Keywords Database

Categories Covered:

1. **Programming Languages** (29): Python, Java, JavaScript, TypeScript, C++, Go, Rust, etc.
2. **Web Frameworks** (27): Django, Flask, React, Angular, Vue, Spring Boot, etc.
3. **Mobile Development** (11): Flutter, React Native, SwiftUI, Android, iOS, etc.
4. **Databases** (28): PostgreSQL, MongoDB, Redis, MySQL, Elasticsearch, etc.
5. **AI/ML/Data Science** (60+): TensorFlow, PyTorch, Transformers, LangChain, etc.
6. **DevOps & Cloud** (40+): AWS, Docker, Kubernetes, Terraform, Jenkins, etc.
7. **Tools & Platforms** (30+): Git, Postman, Figma, VSCode, Jupyter, etc.

8. **Cybersecurity** (25+): Penetration Testing, Burp Suite, Wireshark, etc.
9. **Data Engineering** (20+): Spark, Hadoop, Airflow, Kafka, dbt, etc.
10. **Testing** (15+): Pytest, Selenium, Cypress, Jest, Playwright, etc.

Total: 200+ technologies recognized

---

9. Storage System

9.1 File Structure

```
candidate_data/
├── candidate_20250106_143022.json
├── candidate_20250106_151530.json
└── candidate_20250106_163045.json
```

Naming Convention:

```
filename = f"candidate_{timestamp}.json"
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

9.2 Storage Function

```
def save_candidate_data(
    candidate_info: dict,
    questions: List[str],
    answers: List[str]
) -> str:
    """
    Saves complete candidate screening data to JSON file

    Returns: filename of saved data
    """
    # Generate timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"candidate_data/candidate_{timestamp}.json"

    # Ensure directory exists
    os.makedirs("candidate_data", exist_ok=True)

    # Create Q&A pairs
    qa_pairs = [
        {
            "question_number": i,
            "question": q,
            "answer": a
        }
        for i, (q, a) in enumerate(zip(questions, answers), 1)
    ]

    # Compile complete data
    data = {
        "timestamp": timestamp,
        "candidate_info": candidate_info,
        "technical_questions": questions,
        "candidate_answers": answers,
        "questions_and_answers": qa_pairs,
        "status": "initial_screening_complete"
    }

    # Write to file
    with open(filename, 'w') as f:
        json.dump(data, f, indent=2)

    return filename
```

9.3 Data Retrieval

```
import json

# Load candidate data
with open('candidate_data/candidate_20250106_143022.json', 'r') as f:
    data = json.load(f)

# Access specific fields
name = data['candidate_info']['full_name']
tech_stack = data['candidate_info']['tech_stack']
answers = data['candidate_answers']
```

9.4 Data Migration

For database migration (future enhancement):

```
import json
import sqlite3
```

```
def migrate_to_database():
    conn = sqlite3.connect('candidates.db')
    cursor = conn.cursor()

    # Create tables
    cursor.execute('''
        CREATE TABLE candidates (
            id INTEGER PRIMARY KEY,
            timestamp TEXT,
            full_name TEXT,
            email TEXT UNIQUE,
            phone TEXT,
            years_experience TEXT,
            desired_position TEXT,
            current_location TEXT
        )
    ''')

    # Process JSON files
    for filename in os.listdir('candidate_data'):
        with open(f'candidate_data/{filename}', 'r') as f:
            data = json.load(f)

        # Insert into database
        cursor.execute('''
            INSERT INTO candidates VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            None,  # auto-increment id
            data['timestamp'],
            data['candidate_info']['full_name'],
            data['candidate_info']['email'],
            # ... other fields
        ))

    conn.commit()
    conn.close()
```

---

10. UI Components

10.1 Main Interface

Layout:

```
┌─────────────────────────────────────────────────────────────┐
│ 🎯 TalentScout Hiring Assistant                 │
│ AI-Powered Initial Candidate Screening           │
├─────────────────────────────────────────────────────────────┤
│                                          │
│ [Chat Messages Display Area]                 │
│                                          │
│ Assistant: Welcome! What's your name?            │
│ User: John Doe                               │
│ Assistant: Great! What's your email?           │
│ User: john@example.com                        │
│ ...                                     │
│                                          │
├─────────────────────────────────────────────────────────────┤
│ [Message Input Field]                       │
│ Type your message here...           [Send] │
└─────────────────────────────────────────────────────────────┘
```

10.2 Sidebar Components

```
with st.sidebar:
    # About Section
    st.header("ℹ️ About This Assistant")

    # Session Info
    st.header("📊 Session Info")
    st.metric("Messages Exchanged", len(messages))
    st.metric("Current Field", current_field)

    # Collected Information Display
    st.subheader("✅ Collected Information:")
    for key, value in candidate_info.items():
        st.text(f"{key}: {value}")

    # Reset Button
    st.button("🔄 Start New Session")
```

10.3 Custom Styling

```
st.markdown("""
<style>
    .main-header {
        font-size: 2.5rem;
        color: #1f77b4;
        text-align: center;
        margin-bottom: 1rem;
    }
    .sub-header {
        text-align: center;
        color: #666;
        margin-bottom: 2rem;
    }

    /* Additional custom styles */
    .stChatMessage {
        padding: 1rem;
        border-radius: 0.5rem;
    }
</style>
""", unsafe_allow_html=True)
```

10.4 Message Display

```
# Display all messages
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Chat input
if prompt := st.chat_input("Type your message here..."):
    # Process input
    pass
```

10.5 Loading States

```
with st.spinner("💭 Generating technical questions..."):
    # Perform async operation
    result = asyncio.run(Runner.run(agent, prompt))
```

---

11. API Integration

11.1 Gemini API Configuration

```
# API Endpoint
GEMINI_BASE_URL = "https://generativelanguage.googleapis.com/v1beta/openai/"

# Model Selection
GEMINI_MODEL = "gemini-2.0-flash"

# Client Initialization
gemini_client = AsyncOpenAI(
    base_url=GEMINI_BASE_URL,
    api_key=os.getenv("GEMINI_API_KEY")
)
```

11.2 Request Format

```
# Implicit request via Agent framework
tech_result = asyncio.run(Runner.run(
    tech_question_agent,
    "Generate technical questions for: Python, Django, PostgreSQL"
))
```

Underlying API Call:

```
{
  "model": "gemini-2.0-flash",
  "messages": [
    {
      "role": "system",
      "content": "You are an expert technical interviewer..."
    },
    {
      "role": "user",
      "content": "Generate technical questions for: Python, Django, PostgreSQL"
    }
  ],
  "response_format": {
    "type": "json_schema",
    "json_schema": {
      "name": "TechnicalQuestions",
      "schema": {...}
    }
  }
}
```

11.3 Response Handling

```
# Success case
if tech_result.final_output:
    questions = tech_result.final_output.questions[:5]
    st.session_state.tech_questions = {
        "questions": questions,
        "current_index": 0,
        "answers": []
    }

# Error case (fallback)
else:
    questions = ["Tell me about one project you're most proud of?"]
    # Continue with fallback question
```

11.4 Rate Limiting

Gemini Free Tier Limits:

- 60 requests per minute
- 1,500 requests per day

Handling:

```
try:
    result = asyncio.run(Runner.run(agent, prompt))
except RateLimitError:
    # Fall back to default question
    response = "Could you tell me about one project you're most proud of?"
```

11.5 Error Codes

Code	Meaning	Action
400	Bad Request	Log error, use fallback
401	Invalid API Key	Alert admin
429	Rate Limit	Wait and retry
500	Server Error	Use fallback question

12. Configuration Guide

12.1 Environment Variables

```
# Required
GEMINI_API_KEY=your_gemini_api_key_here

# Optional (with defaults)
GEMINI_MODEL=gemini-2.0-flash
STREAMLIT_SERVER_PORT=8501
LOG_LEVEL=INFO
```

12.2 Model Configuration

```
# Change model
GEMINI_MODEL = "gemini-1.5-pro"  # More accurate but slower
# or
GEMINI_MODEL = "gemini-2.0-flash"  # Faster, cost-effective

# Adjust temperature (creativity)
gemini_model = OpenAIChatCompletionsModel(
    model=GEMINI_MODEL,
    openai_client=gemini_client,
    temperature=0.7  # 0.0 = deterministic, 1.0 = creative
)
```

12.3 Question Configuration

```
# Number of questions per technology
questions = tech_result.final_output.questions[:5]  # Change 5 to desired number

# Total questions across all technologies
max_total_questions = 10  # Add limit if needed
```

12.4 Validation Rules

```
# Customize phone validation
phone_pattern = r"\d{10,15}"  # Change to match your region

# Email validation strictness
email_pattern = r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"

# Name validation
```

```
min_name_words = 1  # Change minimum words in name
max_name_words = 3  # Change maximum words in name
```

12.5 UI Configuration

```
# Page configuration
st.set_page_config(
    page_title="TalentScout Hiring Assistant",
    page_icon="🎯",
    layout="wide",  # or "centered"
    initial_sidebar_state="expanded"  # or "collapsed"
)

# Theme (in .streamlit/config.toml)
[theme]
primaryColor="#1f77b4"
backgroundColor="#ffffff"
secondaryBackgroundColor="#f0f2f6"
textColor="#262730"
font="sans serif"
```

---

13. Extending the System

13.1 Adding New Fields

Step 1: Update Field Order

```
field_order = [
    "full_name", "email", "phone",
    "years_experience", "desired_position",
    "current_location", "tech_stack",
    "linkedin_profile"  # NEW FIELD
]
```

Step 2: Add Validation Function

```
elif field == "linkedin_profile":
    pattern = r"https://www\.linkedin\.com/in/[\w-]+"
    if re.match(pattern, content):
        extracted["linkedin_profile"] = content
```

Step 3: Add Prompts

```
prompts = {
    # ... existing prompts ...
    "current_location": "Great! Where are you currently located?",
    "tech_stack": "Thank you! Could you list your main technologies?",
    "linkedin_profile": "Perfect! What's your LinkedIn profile URL?"  # NEW
}

clarifications = {
    # ... existing clarifications ...
    "linkedin_profile": "Please provide your LinkedIn URL (e.g.,
https://www.linkedin.com/in/yourname)"
}
```

Step 4: Update Data Model

```
class CandidateInfo(BaseModel):
    full_name: Optional[str] = None
    email: Optional[str] = None
    phone: Optional[str] = None
    years_experience: Optional[str] = None
    desired_position: Optional[str] = None
    current_location: Optional[str] = None
    tech_stack: Optional[List[str]] = None
    linkedin_profile: Optional[str] = None  # NEW
```

13.2 Creating Custom Agents

Example: Resume Analysis Agent

```
# Define output model
class ResumeAnalysis(BaseModel):
    skills_match_score: int = Field(description="0-100 score")
    key_strengths: List[str] = Field(description="Top 3 strengths")
    areas_for_improvement: List[str] = Field(description="Areas to improve")
    recommendation: str = Field(description="Hire/No Hire/Maybe")

# Create agent
resume_analyzer_agent = Agent(
    name="Resume Analyzer",
    instructions="""
    You are an expert resume reviewer for technical positions.
    Analyze the candidate's responses and provide:
    1. Skills match score (0-100)
    2. Top 3 key strengths
    3. Areas for improvement
    4. Final recommendation

    Be objective, fair, and constructive.
    """,
    model=gemini_model,
    output_type=ResumeAnalysis,
)

# Use the agent
analysis = asyncio.run(Runner.run(
    resume_analyzer_agent,
    f"Analyze this candidate: {candidate_info}"
))
```

13.3 Adding Multiple AI Models

```
# OpenAI GPT-4
from openai import AsyncOpenAI as OpenAIAsync

openai_client = OpenAIAsync(api_key=os.getenv("OPENAI_API_KEY"))
gpt4_model = OpenAIChatCompletionsModel(
    model="gpt-4-turbo",
    openai_client=openai_client
)

# Use different models for different tasks
tech_question_agent = Agent(
    name="Technical Questions",
    model=gemini_model,  # Fast, cost-effective
    # ...
)

resume_analyzer_agent = Agent(
    name="Resume Analyzer",
    model=gpt4_model,  # More accurate analysis
    # ...
)
```

13.4 Integrating External Services

Email Notification Service

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

def send_notification(candidate_email: str, candidate_name: str):
    """Send confirmation email to candidate"""

    sender = "noreply@talentscout.com"
    password = os.getenv("EMAIL_PASSWORD")

    message = MIMEMultipart()
    message["From"] = sender
    message["To"] = candidate_email
    message["Subject"] = "TalentScout - Application Received"

    body = f"""
    Dear {candidate_name},

    Thank you for completing the initial screening with TalentScout!

    Our team will review your profile and contact you within 3-5 business days.

    Best regards,
    TalentScout Team
    """

    message.attach(MIMEText(body, "plain"))

    with smtplib.SMTP_SSL("smtp.gmail.com", 465) as server:
        server.login(sender, password)
        server.send_message(message)

# Call after saving candidate data
save_candidate_data(candidate_info, questions, answers)
send_notification(
    candidate_info["email"],
    candidate_info["full_name"]
)
```

Database Integration

```
import psycopg2
from psycopg2.extras import Json

def save_to_database(candidate_info: dict, questions: list, answers: list):
    """Save candidate data to PostgreSQL database"""

    conn = psycopg2.connect(
        host=os.getenv("DB_HOST"),
        database=os.getenv("DB_NAME"),
        user=os.getenv("DB_USER"),
        password=os.getenv("DB_PASSWORD")
    )

    cursor = conn.cursor()

    cursor.execute("""
        INSERT INTO candidates
        (name, email, phone, experience, position, location, tech_stack,
         questions, answers, created_at)
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, NOW())
    """, (
        candidate_info["full_name"],
        candidate_info["email"],
        candidate_info["phone"],
        candidate_info["years_experience"],
        candidate_info["desired_position"],
        candidate_info["current_location"],
        Json(candidate_info["tech_stack"]),
        Json(questions),
        Json(answers)
    ))

    conn.commit()
    cursor.close()
    conn.close()
```

13.5 Multi-Language Support

```
# translations.py
TRANSLATIONS = {
    "en": {
        "greeting": "Hello! Welcome to TalentScout!",
        "ask_name": "What's your full name?",
        "ask_email": "Could you share your email address?",
        # ... more translations
    },
    "es": {
        "greeting": "¡Hola! ¡Bienvenido a TalentScout!",
        "ask_name": "¿Cuál es tu nombre completo?",
        "ask_email": "¿Podrías compartir tu dirección de correo?",
        # ... more translations
    },
    "fr": {
        "greeting": "Bonjour! Bienvenue à TalentScout!",
        "ask_name": "Quel est votre nom complet?",
        "ask_email": "Pourriez-vous partager votre adresse e-mail?",
        # ... more translations
    }
}

# In app.py
def get_text(key: str, lang: str = "en") -> str:
    return TRANSLATIONS.get(lang, TRANSLATIONS["en"]).get(key, key)

# Usage
st.session_state.language = "en"  # or detect from user
greeting = get_text("greeting", st.session_state.language)
```

---

14. Troubleshooting

14.1 Common Issues and Solutions

Issue: "GEMINI_API_KEY is not set"

Symptoms:

```
RuntimeError: ❌ GEMINI_API_KEY is not set in environment variables.
```

Solutions:

1. Verify `.env` file exists in project root

Check `.env` content:

```
cat .env# Should show: GEMINI_API_KEY=your_key_here
```

2.

Ensure no extra spaces:
 # WrongGEMINI_API_KEY = your_key# CorrectGEMINI_API_KEY=your_key

3.
4. Restart Streamlit after updating `.env`

Issue: Session State Resets Unexpectedly

Symptoms:

- Conversation restarts mid-flow
- Collected data disappears

Solutions:

```
# Add session state persistence
if "initialized" not in st.session_state:
    st.session_state.initialized = True
    # Initialize only once

# Debug session state
st.sidebar.write("Debug:", st.session_state)
```

Issue: Agent Returns No Output

Symptoms:

```
tech_result.final_output is None
```

Solutions:

1. Check API key validity
2. Verify internet connection
3. Implement robust fallback:

```
try:
    tech_result = asyncio.run(Runner.run(agent, prompt))
    if tech_result.final_output:
        questions = tech_result.final_output.questions
    else:
        raise ValueError("Empty response")
except Exception as e:
    logging.error(f"Agent error: {e}")
    questions = DEFAULT_QUESTIONS
```

Issue: Validation Not Working

Symptoms:

- Invalid data accepted
- Valid data rejected

Solutions:

```
# Debug validation
extracted = extract_candidate_info_last_message(prompt, field)
st.write(f"DEBUG: Extracted: {extracted}")
st.write(f"DEBUG: Field: {field}")
st.write(f"DEBUG: Prompt: {prompt}")
```

Issue: Tech Stack Not Recognized

Symptoms:

- Technologies not extracted
- Empty tech_stack

Solutions:

1. Check spelling in tech_keywords list
2. Add variations:

```
tech_keywords = [
    "react", "reactjs", "react.js",  # Multiple variations
    "node", "nodejs", "node.js",
    # ...
]
```

3. Implement fuzzy matching:

```
from difflib import get_close_matches

matched_techs = []
for word in lower_content.split():
    matches = get_close_matches(word, tech_keywords, n=1, cutoff=0.8)
    matched_techs.extend(matches)
```

14.2 Debugging Techniques

Enable Logging

```
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('app.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# Use throughout code
logger.debug(f"User input: {prompt}")
logger.info(f"Field transition: {old_field} -> {new_field}")
logger.error(f"Validation failed: {error}")
```

Session State Inspector

```
# Add to sidebar
with st.sidebar:
    with st.expander("🔍 Debug Info"):
        st.write("Session State:", st.session_state)
        st.write("Current Field:", st.session_state.current_field)
        st.write("Candidate Info:", st.session_state.candidate_info)
```

Network Debugging

```
import requests

# Test Gemini API connectivity
def test_api_connection():
    try:
        response = requests.get(
            "https://generativelanguage.googleapis.com",
            timeout=5
        )
        return True
    except Exception as e:
        st.error(f"API connection failed: {e}")
        return False

# Call before starting conversation
if not test_api_connection():
    st.error("Cannot connect to Gemini API. Check your internet connection.")
    st.stop()
```

14.3 Performance Issues

Slow Response Times

Solutions:

1. **Use faster model:**

```
GEMINI_MODEL = "gemini-2.0-flash"  # Faster
# vs
GEMINI_MODEL = "gemini-1.5-pro"  # More accurate but slower
```

2. **Implement caching:**

```
@st.cache_data(ttl=3600)
def generate_questions_cached(tech_stack: str):
    return asyncio.run(Runner.run(agent, tech_stack))
```

3. **Parallel processing:**

```
import concurrent.futures

async def generate_multiple_questions(tech_stacks: list):
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [
            executor.submit(asyncio.run, Runner.run(agent, tech))
            for tech in tech_stacks
        ]
        results = [f.result() for f in futures]
    return results
```

Memory Issues

Solutions:

1. **Limit message history:**

```
# Keep only last 20 messages
if len(st.session_state.messages) > 20:
    st.session_state.messages = st.session_state.messages[-20:]
```

2. **Clear old sessions:**

```
# Auto-clear after 1 hour of inactivity
import time

if "last_activity" not in st.session_state:
    st.session_state.last_activity = time.time()

if time.time() - st.session_state.last_activity > 3600:
    # Clear session
    for key in list(st.session_state.keys()):
        del st.session_state[key]
```

---

15. Best Practices

15.1 Code Organization

Recommended Structure:

```
talentscout/
├── app.py                  # Main application
├── agents/
│   ├── __init__.py
│   ├── question_generator.py  # Question generation agent
│   └── resume_analyzer.py     # Resume analysis agent
├── models/
│   ├── __init__.py
│   └── data_models.py         # Pydantic models
├── utils/
│   ├── __init__.py
│   ├── validation.py          # Validation functions
│   ├── storage.py             # Data storage functions
│   └── helpers.py             # Helper functions
│
```

```
├── config/
│   ├── __init__.py
│   └── settings.py          # Configuration
├── tests/
│   ├── test_validation.py
│   ├── test_agents.py
│   └── test_storage.py
├── .env
├── .gitignore
├── requirements.txt
├── README.md
└── DOCUMENTATION.md
```

15.2 Error Handling

Always use try-except blocks:

```
try:
    # Risky operation
    result = asyncio.run(Runner.run(agent, prompt))
except TimeoutError:
    logger.error("Agent timeout")
    result = fallback_result
except APIError as e:
    logger.error(f"API error: {e}")
    result = fallback_result
except Exception as e:
    logger.error(f"Unexpected error: {e}")
    result = fallback_result
```

User-friendly error messages:

```
# Bad
st.error("Error: NoneType object has no attribute 'questions'")

# Good
st.error("We're having trouble generating questions. Using default questions instead.")
```

15.3 Security Practices

Never commit sensitive data:

```
# .gitignore
.env
*.json
candidate_data/
__pycache__/
*.pyc
.venv/
venv/
```

Sanitize user input:

```python
import html

def sanitize_input(user_input: str) -> str:
    """Sanitize user input to prevent XSS"""
    return html.escape(user_input.strip())

# Use before storing
prompt = sanitize_input(user_prompt)
```

Validate before API calls:

```python
def validate_before_api_call(tech_stack: list) -> bool:
    """Validate data before sending to API"""
    if not tech_stack:
        return False
    if len(tech_stack) > 20:  # Limit to prevent abuse
        return False
    return True
```

15.4 User Experience

Provide clear feedback:

```python
with st.spinner("💭 Generating personalized questions for you..."):
    questions = generate_questions(tech_stack)

st.success("✅ Questions generated! Let's begin...")
```

Handle long waits:

```
import time

with st.spinner("Processing..."):
    start = time.time()
    result = long_operation()
    elapsed = time.time() - start

    if elapsed > 5:
        st.info(f"That took {elapsed:.1f} seconds. Thanks for your patience!")
```

Progressive disclosure:

```
# Don't show all questions at once
current_q = st.session_state.tech_questions["current_index"]
total = len(st.session_state.tech_questions["questions"])

st.progress(current_q / total)
st.caption(f"Question {current_q + 1} of {total}")
```

15.5 Testing

Unit Tests:

```
# tests/test_validation.py
import pytest
from utils.validation import extract_candidate_info_last_message

def test_email_validation():
    # Valid email
    result = extract_candidate_info_last_message("john@example.com", "email")
    assert "email" in result
    assert result["email"] == "john@example.com"

    # Invalid email
    result = extract_candidate_info_last_message("invalid-email", "email")
    assert "email" not in result

def test_phone_validation():
    # Valid phone
    result = extract_candidate_info_last_message("1234567890", "phone")
    assert "phone" in result

    # Invalid phone (too short)
    result = extract_candidate_info_last_message("12345", "phone")
    assert "phone" not in result
```

Integration Tests:

```
# tests/test_agents.py
import pytest
import asyncio
from agents.question_generator import tech_question_agent

@pytest.mark.asyncio
async def test_question_generation():
    result = await Runner.run(
        tech_question_agent,
        "Generate questions for: Python, Django"
    )

    assert result.final_output is not None
    assert len(result.final_output.questions) > 0
    assert isinstance(result.final_output.questions, list)
```

---

16. Security & Privacy

16.1 Data Protection

GDPR Compliance:

```
# Add data retention policy
DATA_RETENTION_DAYS = 90

def cleanup_old_data():
    """Delete candidate data older than retention period"""
    import os
    from datetime import datetime, timedelta

    cutoff_date = datetime.now() - timedelta(days=DATA_RETENTION_DAYS)

    for filename in os.listdir("candidate_data"):
        filepath = os.path.join("candidate_data", filename)
        file_time = datetime.fromtimestamp(os.path.getmtime(filepath))
        if file_time < cutoff_date:
            os.remove(filepath)
            logger.info(f"Deleted old file: {filename}")

# Run periodically
cleanup_old_data()
```

Data Minimization:

```
# Only collect necessary data
class CandidateInfo(BaseModel):
    # Required fields only
    full_name: str
    email: str
    tech_stack: List[str]

    # Optional fields
    phone: Optional[str] = None
    # Don't collect: SSN, date of birth, etc.
```

Consent Management:

```
# Add consent checkbox
if not st.session_state.get("consent_given"):
    st.warning("⚠️ Data Privacy Notice")

    consent = st.checkbox("""
    I consent to TalentScout storing my personal information
    for recruitment purposes. Data will be retained for 90 days.
    """)

    if consent:
        st.session_state.consent_given = True
        st.rerun()
    else:
        st.stop()
```

16.2 API Security

Rate Limiting:

```
from collections import defaultdict
from datetime import datetime, timedelta

class RateLimiter:
    def __init__(self, max_requests=60, window_seconds=60):
        self.max_requests = max_requests
        self.window = timedelta(seconds=window_seconds)
        self.requests = defaultdict(list)

    def is_allowed(self, user_id: str) -> bool:
        now = datetime.now()
        cutoff = now - self.window

        # Clean old requests
        self.requests[user_id] = [
            req_time for req_time in self.requests[user_id]
            if req_time > cutoff
        ]

        # Check limit
        if len(self.requests[user_id]) >= self.max_requests:
            return False

        self.requests[user_id].append(now)
        return True

rate_limiter = RateLimiter()

# Use before API calls
if not rate_limiter.is_allowed(session_id):
    st.error("Too many requests. Please wait a moment.")
    st.stop()
```

Input Sanitization:

```
import re

def sanitize_for_api(text: str) -> str:
    """Clean text before sending to API"""
    # Remove control characters
    text = re.sub(r'[\x00-\x1f\x7f-\x9f]', '', text)

    # Limit length
    max_length = 1000
    text = text[:max_length]
    # Remove potentially malicious patterns
    text = re.sub(r'<script.*?</script>', '', text, flags=re.DOTALL)

    return text.strip()
```

16.3 Access Control

Admin Dashboard (Future Enhancement):

```
# Implement authentication
import streamlit_authenticator as stauth

def require_auth():
    """Require authentication for admin features"""
    authenticator = stauth.Authenticate(
        credentials,
        cookie_name='talentscout_auth',
        key='auth_key',
        cookie_expiry_days=30
    )

    name, authentication_status, username = authenticator.login('Login', 'main')

    if not authentication_status:
        st.stop()

    return username

# Protect admin routes
if st.sidebar.checkbox("Admin Mode"):
    username = require_auth()
    # Show admin features
```

---

17. Performance Optimization

17.1 Caching Strategies

Cache AI Responses:

```
@st.cache_data(ttl=3600, show_spinner=False)
```

```
def generate_questions_for_tech(tech_stack_str: str):
    """Cache generated questions for 1 hour"""
    return asyncio.run(Runner.run(agent, tech_stack_str))

# Use cached version
tech_stack_key = ",".join(sorted(tech_stack))
questions = generate_questions_for_tech(tech_stack_key)
```

Cache Validation Results:

```
@st.cache_data
def get_tech_keywords():
    """Cache tech keywords list"""
    return [
        "python", "java", "javascript",
        # ... 200+ keywords
    ]

tech_keywords = get_tech_keywords()
```

17.2 Async Optimization

Parallel API Calls:

```
async def generate_multiple_question_sets(tech_groups: list):
    """Generate questions for multiple tech groups in parallel"""
    tasks = [
        Runner.run(tech_question_agent, f"Questions for: {tech}")
        for tech in tech_groups
    ]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    return [r for r in results if not isinstance(r, Exception)]

# Use when candidate has many technologies
if len(tech_stack) > 5:
    # Group technologies
    tech_groups = [tech_stack[i:i+3] for i in range(0, len(tech_stack), 3)]
    all_questions = asyncio.run(generate_multiple_question_sets(tech_groups))
```

17.3 Resource Management

Memory Optimization:

```
# Limit stored data
MAX_MESSAGE_HISTORY = 50

def trim_message_history():
    if len(st.session_state.messages) > MAX_MESSAGE_HISTORY:
        # Keep first (greeting) and last N messages
        st.session_state.messages = (
            [st.session_state.messages[0]] +
            st.session_state.messages[-MAX_MESSAGE_HISTORY:]
        )

# Call after each interaction
trim_message_history()
```

File Storage Optimization:

```
# Compress JSON files
import gzip
import json

def save_compressed(data: dict, filename: str):
    with gzip.open(f"{filename}.gz", 'wt', encoding='utf-8') as f:
        json.dump(data, f, indent=2)

# Reduces storage by ~70%
save_compressed(candidate_data, f"candidate_{timestamp}.json")
```

---

18. Testing

18.1 Test Structure

```
tests/
├── __init__.py
├── test_validation.py
├── test_agents.py
├── test_storage.py
├── test_integration.py
└── conftest.py  # Pytest fixtures
```

18.2 Sample Tests

conftest.py:

```
import pytest
from unittest.mock import Mock

@pytest.fixture
def sample_candidate_info():
    return {
        "full_name": "John Doe",
        "email": "john@example.com",
        "phone": "1234567890",
        "years_experience": "5",
        "desired_position": "Software Engineer",
        "current_location": "New York, USA",
        "tech_stack": ["python", "django", "postgresql"]
    }

@pytest.fixture
def mock_agent():
    agent = Mock()
    agent.run.return_value = Mock(
        final_output=Mock(
            questions=["Question 1?", "Question 2?"],
            tech_category="Python"
        )
    )
    return agent
```

test_validation.py:

```
import pytest
from utils.validation import (
    extract_candidate_info_last_message,
    is_exit_keyword
)

class TestValidation:
    def test_valid_email(self):
        result = extract_candidate_info_last_message(
            "john@example.com",
            "email"
        )
        assert result == {"email": "john@example.com"}
    def test_invalid_email(self):
        result = extract_candidate_info_last_message(
            "not-an-email",
            "email"
        )
        assert result == {}

    def test_exit_keywords(self):
        assert is_exit_keyword("bye")
        assert is_exit_keyword("EXIT")
        assert is_exit_keyword("Goodbye!")
        assert not is_exit_keyword("hello")

    def test_tech_stack_extraction(self):
        result = extract_candidate_info_last_message(
            "I know Python, Django, and React",
            "tech_stack"
        )
        assert "tech_stack" in result
        assert "python" in result["tech_stack"]
        assert "django" in result["tech_stack"]
```

Run tests:

```
# Run all tests
pytest

# Run with coverage
pytest --cov=. --cov-report=html

# Run specific test file
pytest tests/test_validation.py

# Run with verbose output
pytest -v
```

---

19. Deployment

19.1 Streamlit Cloud Deployment

Step 1: Prepare repository

```
# Ensure these files exist:
# - app.py
# - requirements.txt
# - .streamlit/config.toml (optional)

git init
git add .
git commit -m "Initial commit"
git remote add origin <your-repo-url>
git push -u origin main
```

Step 2: Deploy on Streamlit Cloud

1. Go to share.streamlit.io
2. Click "New app"
3. Select repository
4. Set main file: `app.py`
5. Click "Deploy"

Step 3: Add secrets In Streamlit Cloud dashboard:

```
# Secrets section
GEMINI_API_KEY = "your_api_key_here"
```

19.2 Docker Deployment

Dockerfile:

```
FROM python:3.10-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .

# Expose Streamlit port
EXPOSE 8501

# Health check
```

```
HEALTHCHECK CMD curl --fail http://localhost:8501/_stcore/health

# Run application
ENTRYPOINT ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

docker-compose.yml:

```
version: '3.8'

services:
  talentscout:
    build: .
    ports:
      - "8501:8501"
    environment:
      - GEMINI_API_KEY=${GEMINI_API_KEY}
    volumes:
      - ./candidate_data:/app/candidate_data
    restart: unless-stopped
```

Deploy:

```
# Build image
docker build -t talentscout .

# Run container
docker run -p 8501:8501 \
  -e GEMINI_API_KEY=your_key \
  -v $(pwd)/candidate_data:/app/candidate_data \
  talentscout

# Or use docker-compose
docker-compose up -d
```

19.3 AWS Deployment

Using AWS ECS:

```
# 1. Push to ECR
aws ecr create-repository --repository-name talentscout
docker tag talentscout:latest <account-id>.dkr.ecr.<region>.amazonaws.com/talentscout:latest
docker push <account-id>.dkr.ecr.<region>.amazonaws.com/talentscout:latest
```

```
# 2. Create ECS task definition
# 3. Create ECS service
# 4. Configure load balancer
```

19.4 Production Checklist

- [ ] Environment variables configured
- [ ] HTTPS enabled
- [ ] Rate limiting implemented
- [ ] Monitoring setup
- [ ] Backup strategy defined
- [ ] Error tracking enabled (e.g., Sentry)
- [ ] Load testing completed
- [ ] Security audit performed
- [ ] Documentation updated
- [ ] Team training completed

---

20. FAQ

Q1: Can I use a different AI model?

A: Yes! The system supports any OpenAI-compatible API:

```
# Use OpenAI GPT-4
openai_client = AsyncOpenAI(api_key=os.getenv("OPENAI_API_KEY"))
model = OpenAIChatCompletionsModel(
    model="gpt-4-turbo",
    openai_client=openai_client
)

# Use Anthropic Claude
anthropic_client = AsyncAnthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
# Configure accordingly
```

Q2: How do I handle multiple concurrent users?

A: Streamlit handles each user session separately. For high traffic:

1. Deploy multiple instances

2. Use load balancer
3. Consider caching shared data
4. Implement connection pooling

Q3: Can I integrate with an ATS (Applicant Tracking System)?

A: Yes! Add webhook integration:

```
import requests

def send_to_ats(candidate_data: dict):
    """Send candidate data to ATS via API"""
    ats_webhook_url = os.getenv("ATS_WEBHOOK_URL")

    response = requests.post(
        ats_webhook_url,
        json=candidate_data,
        headers
```