이루다 프로젝트 발표 스크립트

오프닝 (슬라이드 1)

안녕하세요. 오늘 저희가 소개할 프로젝트는 **"이루다"**입니다.
기술과 따뜻함이 만나는 새로운 자립 지원 시스템, 자립청소년을 위한 AI 컨설팅 Agent를 발표하겠습니다.

저희는 세이브 더 칠드런 소속 배정윤, 이상진, 김준희, 이상엽 팀입니다.

Chapter 1: 현황 및 문제 인식 (슬라이드 3-8)

자립이 필요한 청소년의 현실

현재 우리나라에는 **1만 명 이상의 보호종료청소년**이 있습니다. 이들은 경제적 자립, 주거 안정, 심리적 지원 등 다방면에서 심각한 어려움에 직면하고 있습니다.

주요 문제점들

2023년 자립준비청년 패널조사 결과를 보면:

- **생활, 학비 등 필요한 돈의 부족**: 19.6%
- **거주할 집 문제**: 17.3%
- **취업에 필요한 정보, 기술, 자격 부족**: 14.9%

이처럼 자립준비청년들은 홀로서기를 시작하며 복합적인 어려움에 직면합니다.

Kosis 통계로 본 수요 예측

보호종료 청소년들이 경험한 자립교육 프로그램의 도움률을 보면:

- **주거**: 79.3%
- **경제**: 79.1%
- **심리**: 78.8%

AI 컨설팅 Agent에 대한 수요 예측도 비슷한 수치를 보여줍니다.

기존 지원 시스템의 한계

현재 복지로 등의 기존 시스템은 여러 한계가 있습니다:

복지로 챗봇의 문제점:

- 단순한 키워드 매칭 방식
- 개인화된 상담 부족
- 복잡한 검색 과정

사회복지사 인터뷰 결과:

- "통합된 신청 루트가 없다"
- "지원 프로그램마다 조건이 제각각이다"
- "기관을 통한 신청시, 기관의 추천서가 필요하다"
- "자립대상 청소년은 지원 정책 용어의 의미를 모른다"
- "제한된 자원인 사회복지사가 모든 자립 대상 청소년을 위한 상담에 쏟을 물리적 시간이 부족하다"

Chapter 2: 프로젝트 '이루다' 소개 (슬라이드 9-12)

이루다의 비전

이러한 문제들을 해결하기 위해 **'AI 컨설팅 Agent: 이루다'**를 제안합니다.

이루다는 정보의 격차를 해소하고 자립의 모든 과정을 따뜻하게 지원하는 든든한 동반자입니다.

핵심 기능

- **개인화된 로드맵 제공**
- **복잡한 정책 및 제도 자동 연계**
- **생활, 교육, 취업, 심리까지 아우르는 통합 지원**

서비스 확장 로드맵

미래에는 다음과 같은 확장이 가능합니다:

1. **다국어 지원**: 외국인 및 다문화 가정 청소년을 위한 언어 장벽 없는 지원
2. **음성 인터페이스**: 디지털 기기 사용이 어려운 사용자를 위한 챗봇 및 음성 인식 기능
3. **민간·기업 연계**: 기업과의 협력을 통한 인턴십, 취업, 전문 멘토링 프로그램 자동 매칭
4. **금융 교육 모듈**: 예산 관리, 저축, 신용 관리 등 건강한 경제 관념 교육

기대 효과

- **청소년 안전망 강화**: 긴급 주거, 의료, 심리 지원이 필요한 위기 상황 발생시 즉각적 연결
- **데이터 분석 리포트**: 축적된 데이터를 익명으로 분석하여 실효성 있는 청년 정책 수립 근거 제공
- **다채널 접근성 확대**: 모바일 앱과 공공장소 오프라인 키오스크 설치

Chapter 3: MVP 목표 (슬라이드 13-14)

핵심 기능 4가지

초기 버전은 다음 4가지 핵심 기능을 중심으로 개발합니다:

1. **사용자 등록 및 기본 정보 입력**: 간편한 가입 절차와 필수 정보 입력

2. **AI 기반 개인화 로드맵 생성**: 경제, 주거, 심리 영역별 추천을 담은 맞춤형 계획 제공
3. **정책·제도 매칭 및 신청 가이드**: 사용자에게 적합한 지원 정책 찾기 및 관련 정보 링크와 신청 양식 제공
4. **진행 상황 저장 및 피드백**: 사용자 진행 상황 기록 및 서비스 개선을 위한 피드백 수집

Chapter 4: 구현 아키텍처 및 워크플로우 (슬라이드 15)

기술 스택

- **Frontend**: React
- **Backend**: FastAPI
- **Database**: PostgreSQL
- **Vector DB**: FAISS
- **AI Model**: GPT-4o (초기 버전)
- **Infrastructure**: Docker, AWS
- **RAG 시스템** 구현

시스템 구조

5개 레이어로 구성된 안정적인 아키텍처:

1. **User Layer**: 보호 청소년
2. **Interface Layer**: React 기반 UI/UX
3. **Application Layer**: FastAPI 백엔드
4. **LLM Layer**: 문장 분석, 정책 검색 모듈, 정책 추천 생성
5. **Data Layer**: 정책 정보 DB, 사용자 DB, Vector DB

Chapter 5: 개발 타임라인 (슬라이드 16-17)

9개월 개발 계획

현재 진행률을 보면:

- **프로젝트 기획**: 90% 완료
- **니즈 분석 및 데이터 수집**: 85% 완료
- **ERD 및 DB 설계**: 40% 완료
- **데이터 정제 및 적재**: 50% 완료
- **LLM 모델 연동 및 파인튜닝**: 65% 완료
- **정책 추천 로직 구현**: 25% 완료
- **사용자 지원 로드맵 기능**: 40% 완료
- **Backend API 개발**: 30% 완료
- **Frontend 개발**: 65% 완료
- **테스트 및 QA**: 0% (예정)

Chapter 6: 개발 현황 (슬라이드 18-46)

데이터 수집 완료

56개 이상의 정책 데이터를 수집하여 Test용 DB를 제작 완료했습니다. 복지로 사이트를 크롤링하여 공공 및 민간 자립지원 정책을 수집했습니다.

평가 방법

RAGAS 프레임워크를 활용한 정확성 평가:

- **Faithfulness**: 답변이 근거 문서에 기반했는가 (할루시네이션 감지)
- **Answer Relevancy**: 답변이 질문과 관련이 있는가 (불필요 답변 감지)
- **Context Precision**: 가져온 문서 중 불필요 문서 비율 (검색 효율성)
- **Context Recall**: 필요한 문서를 얼마나 가져왔는가 (검색 누락 확인)

Mock-Up 구현 현황

실제 작동하는 웹 애플리케이션을 구현했습니다:

주요 페이지들:

1. **홈 및 로그인 페이지**: 사용자 친화적인 인터페이스
2. **회원가입**: 맞춤 지원을 위한 기본 조건 설정 (주거 상황, 소득 수준, 관심 영역 선택)
3. **대시보드**:
   - 진행중인 할 일, 완료된 할 일, 추천 정책, 달성률 표시
   - 최근 활동 및 이번 달 진행 현황 시각화
4. **AI 채팅 인터페이스**: 자연스러운 대화형 상담
5. **자립 로드맵 서비스**: 1개월, 3개월, 6개월 목표별 맞춤형 계획 제공
6. **할 일 관리**: 우선순위별 태스크 관리 시스템
7. **지원정책 바로찾기**: 키워드 검색 및 카테고리별 정책 매칭

기술적 구현 상세 설명

OpenAI API 연동 코드 (슬라이드 30)

```
python
```

```python
def call_openai_api(message, conversation_history=None):
    """OpenAI API를 호출하여 응답을 받아옵니다."""
    global openai_client

    if not openai_client:
        return generate_mock_response(message)

    try:
        # 시스템 메시지 설정 - AI의 역할과 성격을 정의
        system_message = {
            "role": "system",
            "content": f"""당신은 자립준비청년을 위한 AI 상담사입니다.
            친근하고 따뜻하며 실용적인 조언을 제공합니다.

            주요 역할:
            1. 자립준비청년의 고민과 질문에 공감하며 답변
            2. 정부 지원정책과 제도에 대한 정확 정보 제공
            3. 개인별 맞춤 로드맵 및 계획 수립 지원
            4. 주거, 경제, 교육, 취업, 심리 지원 관련 안내

            말투: 친근하면서도 전문적, 격려하고 지지하는 톤
            길이: 3-5문장으로 간결하게, 필요시 구체적인 행동방안 제시"""
        }

        # 대화 히스토리 구성
        messages = [system_message]
        if conversation_history:
            messages.extend(conversation_history[-10:])  # 최근 10개 메시지만 유지

        messages.append({"role": "user", "content": message})

        response = openai_client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=messages,
            temperature=0.7,
            max_tokens=500
        )

        return response.choices[0].message.content

    except Exception as e:
        print(f"OpenAI API 호출 실패: {e}")
        return generate_mock_response(message)
```

코드 설명:

- `system_message`: AI의 성격과 역할을 정의하는 부분입니다. 자립준비청년을 위한 전문 상담사 역할을 하도록 설정했습니다.
- `conversation_history`: 이전 대화 내용을 기억하여 맥락을 유지합니다. 메모리 효율성을 위해 최근 10개만 보관합니다.
- `temperature=0.7`: AI 응답의 창의성 정도를 조절합니다. 0에 가까울수록 일관된 답변, 1에 가까울수록 창의적 답변을 생성합니다.
- `max_tokens=500`: 응답 길이를 제한하여 간결한 답변을 유도합니다.

정책 매칭 알고리즘 (슬라이드 39-43)

1. 임베딩 모델 초기화

```python
```

```python
class EnhancedPolicyMatcher:
    def __init__(self):
        print("정책 매칭 시스템 초기화 중...")

        # 한국어 특화 임베딩 모델 로드
        try:
            self.model = SentenceTransformer('klue/roberta-large')
            print("KLUE RoBERTa 모델 로드 완료")
        except Exception as e:
            print(f"KLUE 모델 로드 실패, 대체 모델 사용: {e}")
            self.model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

        # 정책 데이터 및 임베딩 초기화
        self.policies = []
        self.policy_embeddings = None
        self.policy_texts = []
        self.initialize_policy_embeddings()

        print(f"정책 매칭 시스템 준비 완료 ({len(self.policies)}개 정책)")
```

기술 용어 설명:

- **임베딩(Embedding)**: 텍스트를 컴퓨터가 이해할 수 있는 숫자 벡터로 변환하는 기술입니다. 의미가 비슷한 문장들은 비슷한 벡터값을 가집니다.
- **KLUE RoBERTa**: 한국어에 특화된 언어 이해 모델입니다. 한국어의 문맥과 의미를 정확히 파악할 수 있습니다.
- **SentenceTransformer**: 문장 단위로 임베딩을 생성하는 라이브러리입니다.

2. 정책 데이터 벡터화

```python
```

```python
def initialize_policy_embeddings(self):
    """정책 데이터를 벡터화하여 메모리에 저장"""
    try:
        # 정책 데이터 로드
        self.policies = self.load_government_policies()

        if not self.policies:
            print("경고: 정책 데이터가 없습니다.")
            return

        # 정책별 검색용 텍스트 생성 (벡터 저리 상능 향상)
        self.policy_texts = []
        for policy in self.policies:
            combined_text = self.create_policy_search_text(policy)
            self.policy_texts.append(combined_text)

        # 일괄 벡터화 (배치 처리 상능 향상)
        print("정책 데이터 벡터화 중...")
        self.policy_embeddings = self.model.encode(
            self.policy_texts,
            show_progress_bar=True,
            batch_size=16,
            convert_to_numpy=True
        )

        print(f"정책 임베딩 생성 완료: {self.policy_embeddings.shape}")

    except Exception as e:
        print(f"정책 임베딩 초기화 실패: {e}")
        self.policies = []
        self.policy_embeddings = None
```

주요 포인트:

- `batch_size=16`: 한 번에 16개씩 처리하여 메모리 효율성을 높입니다.
- `convert_to_numpy=True`: 계산 속도를 위해 NumPy 배열로 변환합니다.
- `show_progress_bar=True`: 사용자에게 진행 상황을 보여줍니다.

3. 정책 검색용 텍스트 생성

```python
```

```python
def create_policy_search_text(self, policy):
    """정책 데이터를 검색에 최적화된 텍스트로 변환"""
    text_parts = []

    # 서비스명 (기준자 높음)
    if policy.get('서비스명'):
        text_parts.append(f"서비스: {policy['서비스명']}")
        text_parts.append(policy['서비스명'])  # 중복으로 가중치 증가

    # 지원대상 (중요)
    if policy.get('지원대상'):
        text_parts.append(f"대상: {policy['지원대상']}")

    # 지원내용 (핵심)
    if policy.get('지원내용'):
        text_parts.append(f"내용: {policy['지원내용']}")

    # 기관명
    if policy.get('기관명'):
        text_parts.append(f"기관: {policy['기관명']}")

    # 신청방법
    if policy.get('신청방법'):
        text_parts.append(f"신청: {policy['신청방법']}")

    # 구분 (중앙부처, 지자체, 민간)
    if policy.get('구분'):
        text_parts.append(f"분류: {policy['구분']}")

    return " ".join(text_parts)
```

설계 의도:

- 중요한 필드별로 접두사를 붙여 의미를 명확히 합니다.
- 서비스명을 중복 포함하여 가중치를 높입니다.
- 사용자가 검색할 가능성이 높은 키워드들을 포함합니다.

4. 의미 기반 정책 검색

```python
```

```python
def semantic_search(self, query, user_profile=None, top_k=10):
    """의미적 유사도 기반 정책 검색"""
    if self.policy_embeddings is None:
        print("경고: 정책 임베딩이 없어 기존 방식 사용")
        return self.fallback_to_keyword_search(query)

    try:
        # 1단계: 사용자 쿼리 전처리 및 확장
        enhanced_query = self.enhance_search_query(query, user_profile)

        # 2단계: 쿼리 벡터화
        query_embedding = self.model.encode([enhanced_query])

        # 3단계: 코사인 유사도 계산
        similarities = cosine_similarity(query_embedding, self.policy_embeddings)[0]

        # 4단계: 상위 후보 선택 (더 많이 선택해서 규칙 기반 필터링)
        top_indices = np.argsort(similarities)[::-1][:top_k * 3]

        # 5단계: 후보 정책들에 대해 규칙 기반 검증 및 점수 계산
        candidates = []
        for idx in top_indices:
            if similarities[idx] < 0.1:  # 너무 낮은 유사도는 제외
                continue

            policy = self.policies[idx].copy()
            eligibility = self.check_eligibility(user_profile, policy) if user_profile else None

            # 종합 점수 계산
            combined_score = self.calculate_combined_score(
```

기술 용어 설명:

- **코사인 유사도(Cosine Similarity)**: 두 벡터 사이의 각도를 측정하여 유사도를 계산합니다. 1에 가까울수록 유사합니다.
- **np.argsort()**: NumPy 함수로, 배열을 정렬했을 때의 인덱스를 반환합니다.
- **[::-1]**: 파이썬 슬라이싱으로, 배열을 역순으로 뒤집습니다 (내림차순 정렬).

5. 사용자 적합성 검증

```python
```

```python
def check_eligibility(self, user_profile, policy):
    """사용자 프로필과 정책의 자격 요건 매칭"""
    if not user_profile:
        return {'eligible': True, 'confidence': 0.5, 'reasons': ['프로필 정보 부족']}

    checks = []
    total_weight = 0
    passed_weight = 0

    # 나이 조건 확인
    age_result = self.check_age_requirement(user_profile, policy)
    checks.append(('나이', age_result))
    total_weight += 0.3
    if age_result['passed']:
        passed_weight += 0.3

    # 소득 조건 확인
    income_result = self.check_income_requirement(user_profile, policy)
    checks.append(('소득', income_result))
    total_weight += 0.4
    if income_result['passed']:
        passed_weight += 0.4

    # 특별 조건 확인 (자립준비청년 등)
    special_result = self.check_special_conditions(user_profile, policy)
    checks.append(('특별조건', special_result))
    total_weight += 0.3
    if special_result['passed']:
        passed_weight += 0.3

    confidence = passed_weight / total_weight if total_weight > 0 else 0.5
    eligible = confidence >= 0.7  # 70% 이상 통과해야 자격 있음
```

가중치 설계 이유:

- **소득** (40%): 대부분의 지원 정책에서 가장 중요한 기준
- **나이** (30%): 연령 제한이 있는 정책이 많음
- **특별조건** (30%): 자립준비청년 같은 특수한 자격 요건

사용자 인터페이스 연동

키워드 제안 기능 (슬라이드 32)

```python
```

```
def check_for_page_suggestion(message):
    """메시지를 분석해서 페이지 이동 제안"""
    message_lower = message.lower()

    suggestions = [
        (['roadmap', '로드맵', '계획'], '/roadmap', '로드맵 페이지에서 체계적인 자립 계획을 세워보시겠어요?'),
        (['policy', '정책', '지원', '급여'], '/policies', '지원정책 페이지에서 맞춤 정책을 찾아보시겠어요?'),
        (['todo', '일일', '체크'], '/todos', '할 일 관리 페이지에서 진행상황을 체크해보시겠어요?')
    ]

    for keywords, url, message in suggestions:
        if any(keyword in message_lower for keyword in keywords):
            return {
                'type': 'redirect',
                'url': url,
                'message': message
            }

    return None
```

이 코드는 사용자가 입력한 메시지를 분석해서 적절한 페이지로 안내하는 기능입니다. 예를 들어 "로드맵을 보고 싶어요"라고 하면 로드맵 페이지로 이동을 제안합니다.

다음 주 일정

1. **RAG 멀티턴 적용**: 더욱 직접적인 RAG 강화 및 기존 시스템과 비교분석
2. **모델 고도화**: 리랭킹 과정 고도화를 통한 안정적인 우선순위 제공
3. **UI 편의성 피드백**: 인터페이스 실시간 피드백을 통한 개선
4. **협력 기관 피드백**: 파트너 기관과의 정기 피드백을 통한 현장 요구사항 반영

마무리 (슬라이드 47-48)

우리의 비전

"모든 청년이 자신의 삶을 온전히 이루다(iruda)"

이루다가 그 믿음의 증거가 될 수 있도록 최선을 다하겠습니다.

Q&A

궁금한 점이 있으시면 편하게 질문해 주시기 바랍니다.

발표 시 주의사항 및 추가 설명

강조할 포인트:

1. **실제 수요**: 통계 데이터로 입증된 명확한 사회적 필요
2. **기술적 차별화**: 단순 검색이 아닌 AI 기반 개인화 상담
3. **실질적 진행**: Mock-up까지 구현된 구체적인 개발 현황
4. **확장 가능성**: 다양한 취약계층으로 확장 가능한 플랫폼

예상 질문 및 답변 준비:

1. **"기존 복지로와의 차이점은?"** → 개인화된 AI 상담, 통합적 접근, 사용자 중심 UI
2. **"정확성은 어떻게 보장하나?"** → RAGAS 평가 프레임워크, 전문가 검증, 지속적 학습
3. **"개인정보 보호는?"** → 익명화 처리, 동의 기반 수집, 보안 강화
4. **"지속가능성은?"** → 정부/기관 파트너십, 확장 가능한 비즈니스 모델

발표 팁:

- 각 챕터마다 2-3분씩 배분하여 총 15-20분 내외로 진행
- 통계 수치는 명확하게 강조
- Mock-up 시연 시 핵심 기능에 집중
- 사회적 가치와 기술적 혁신성을 균형있게 어필