로컬 우선 AI 소프트웨어 개발

클라우드 의존 없이 기기에서 직접 실행하는 AI 애플리케이션 설계와 구현

ThakiCloud (Hyojung Han)

로컬 우선 AI 소프트웨어 개발

클라우드 의존 없이 기기에서 직접 실행하는 AI 애플리케이션 설계와 구현

ThakiCloud (Hyojung Han)

Contents

1 왜 로컬 AI인가: 클라우드의 대가를 다시 묻다

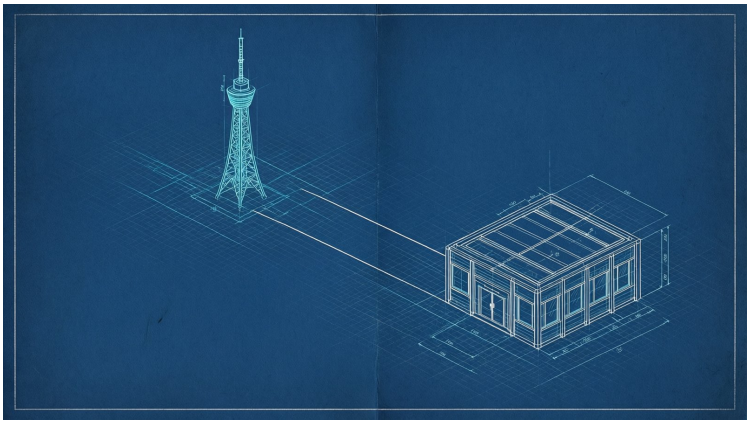

Figure 1.1: 왜 로컬 AI인가: 클라우드의 대가를 다시 묻다

1.1 도입: 구름 너머의 대가

AI 기능을 앱에 넣으려는 개발자 대부분은 먼저 클라우드 API를 고른다. API 키 몇 개만 있으면 며칠 만에 제품을 출시할 수 있다. 하지만 이 선택에는 겉으로 드러나지 않는 비용이 따라붙는다. 청구서에 찍히는 금액도 있지만, 더 무거운 비용은 **데이터 주권**과 **사용자 신뢰**다.

1.2 데이터 주권 침해 실례

클라우드 AI 서비스에 프롬프트를 보내는 순간 입력 데이터는 서비스 제공자의 서버로 넘어간다. 개발사 사정으로 데이터가 저장되거나 보안 사고로 유출되거나 사업 전략이 바뀌어 서비스가 종료되면, 사용자의

데이터는 되돌릴 수 없는 상태에 놓인다. 의료나 금융, 교육처럼 민감한 도메인의 앱이라면 이런 위험은 단순한 번거로움을 넘어 **법적 책임**으로 이어진다.

개인정보보호법과 GDPR이 강화되는 지금, "서버로 데이터를 보내야만 AI가 동작한다"는 구조는 점점 부담스러워진다. 앱의 핵심 기능이 네트워크 연결에 묶여 있으면 오프라인 상태에서는 앱이 통째로 무용지물이 된다. 지하철 안에서든 비행기 안에서든, 해외 로밍 요금이 걱정되는 순간에도 사용자는 여전히 AI 기능이 작동하길 바란다.

1.3 응답 지연의 실질 비용

클라우드 AI의 응답 속도는 네트워크 상태에 크게 좌우된다. 서울에서 미국 서부 리전까지 왕복하는 데만 약 100~150ms가 걸린다. 여기에 실제 모델 추론 시간까지 더해지면 사용자가 체감하는 지연은 1초를 넘기기도 한다.

사용자 경험 연구에 따르면 응답이 300ms를 넘어가는 순간 상호작용의 흐름이 끊긴다. AI 챗봇이라면 답변이 느리게 나타나는 것만으로도 답답하게 느껴질 수 있다. 실시간 음성 대화가 목적이라면 이 지연은 더 치명적이다. 음성 AI가 음성을 텍스트로 바꾸고 텍스트를 처리하고 다시 음성을 합성하는 동안 사용자는 침묵을 견뎌야 한다.

반면 로컬 추론에는 네트워크 왕복이 없다. 적합한 하드웨어에서 돌아가는 작은 모델은 50ms 안에 응답을 만들어낸다. 눈에 띄는 지연 없이 실시간 대화가 가능해지는 이유다.

1.4 서버 비용 구조 분석

클라우드 GPU 인스턴스 비용을 보면 답이 나온다. NVIDIA A100 40GB 인스턴스는 시간당 약 3달러부터 시작한다. 소규모 앱이라도 월간 추론 호출이 수백만 건에 이르면 비용은 빠르게 불어난다. 사용자가 늘어나는

만큼 인프라를 확장하면 비용은 선형이 아니라 지수적으로 증가하는 경향을 보인다.

반면 사용자 디바이스의 GPU는 이미 값을 치른 자산이다. 애플 실리콘의 Neural Engine, 퀄컴 Hexagon DSP, 엔비디아 RTX 시리즈의 CUDA 코어는 유휴 상태일 때의 기회비용은 있어도 사용량에 따라 과금되지는 않는다. 로컬 AI로 아키텍처를 짜면 인프라 비용을 **고정비용에서 변동비용으로** 바꿀 수 있다.

물론 디바이스마다 하드웨어 성능 차이가 크다는 제약은 남는다. 모든 사용자가 고사양 GPU를 갖고 있다고 가정할 수는 없다. 그래서 로컬 AI를 설계할 때는 **폭넓은 하드웨어를 지원하는 유연한 추론 엔진 선택**이 중요해진다.

1.5 로컬 우선 아키텍처의 세 가지 이점

로컬 우선 설계는 핵심 가치 세 가지를 동시에 안겨준다.

첫째는 **데이터 프라이버시**다. 사용자 데이터가 디바이스를 벗어나지 않으니 개인정보보호법을 지키는 부담이 줄어든다. 민감한 내용을 다루는 앱일수록 이 가치는 더 커진다.

둘째는 **즉각적 응답**이다. 네트워크 지연이 없는 로컬 추론은 언제나 일정한 응답 속도를 낸다. 음성이나 영상 인식처럼 지연이 곧바로 체감되는 도메인에서는 이것이 사용자 경험을 가르는 핵심 요소가 된다.

셋째는 **비용 효율성**이다. 서버 인프라를 유지보수할 필요가 없으니 운영 비용이 크게 줄어든다. 사용자가 늘어나도 서버 비용이 그만큼 늘지 않는다.

1.6 이 책의 방향

이 책은 로컬 AI를 실제 제품에 붙이려는 개발자를 위해 썼다. 추론 엔진 선정부터 모델 최적화, 하이브리드 아키텍처 설계, 실제 코드 구현까지 단계별로 다룬다. 독자가 자신의 앱에 맞는 로컬 AI 아키텍처를 직접 설계하고 구현할 수 있도록 구체적인 프레임워크와 단계별 가이드를 담았다.

로컬 AI는 만능 해법이 아니다. 클라우드에 비해 추론 능력이 제한적이고 하드웨어에 따라 성능 편차도 크다. 그럼에도 이 제약 안에서 최고의 사용자 경험을 설계하는 것이 이 책의 목표다.

2 온디바이스 추론 엔진 기술 비교

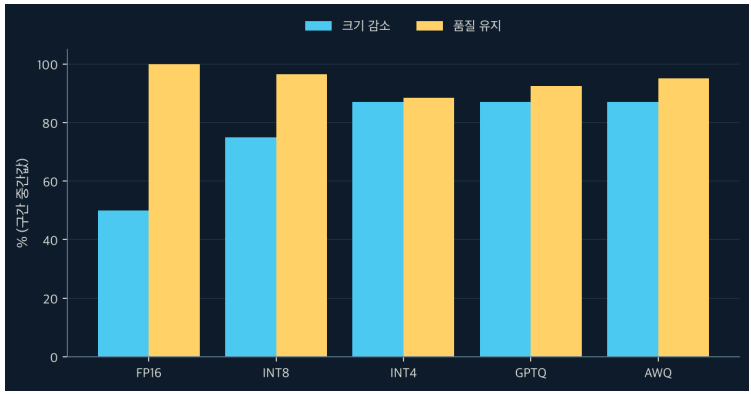

Figure 2.1: 온디바이스 추론 엔진 기술 비교

2.1 추론 엔진 선택의 중요성

로컬 AI 개발에서 가장 중요한 결정 중 하나는 어떤 추론 엔진을 쓸 것인가다. 엔진 선택은 추론 속도, 메모리 사용량, 지원 모델 형식, 플랫폼 호환성에 직접 영향을 준다. 여러 엔진의 특성을 파악하고 프로젝트 요구사항에 맞게 고르는 일이 로컬 AI 아키텍처의 출발점이다.

2.2 주요 엔진 해부

2.2.1 Llama.cpp

Llama.cpp는 C/C++로 작성된 고성능 추론 엔진이며, GGUF 형식의 양자화 모델을 지원한다. CPU 추론에 최적화되어 있고 다양한 플랫폼에서 동작한다. 메탈(Metal) 가속을 쓰면 맥킨토시에서도 GPU로 추론 속도를 끌어올릴 수 있다.

강점은 광범위한 모델 지원과 활발한 커뮤니티다. Llama, Mistral, Qwen 등 대부분의 오픈소스 LLM을 지원한다. 양자화 도구가 잘 갖춰져 있어 INT4, INT8 등 다양한 양자화 수준을 쉽게 시도할 수 있다. CPU 기반 실행이 기본이라 별도의 GPU 드라이버 설정 없이 바로 실행할 수 있다는 점도 실용적이다.

약점은 GPU 활용이다. CUDA 가속은 지원하지만 NVIDIA 특화 최적화는 다른 엔진 대비 부족하다. 대용량 모델을 순수 CPU로 실행하면 속도가 느려질 수 있다.

2.2.2 MNN

MNN은 알리바바가 개발한 경량 추론 엔진이다. 모바일 디바이스에 최적화되어 있어 iOS와 Android에서 뛰어난 성능을 낸다. 모델 변환 도구가 우수하며 다양한 딥러닝 프레임워크와 호환된다.

MNN의 강점은 모바일 플랫폼에 최적화된 성능이다. ARM GPU 가속을 잘 지원하고 메모리 관리 기능도 우수하다. 모델 압축 도구가 내장되어 있어 배포 전 양자화를 간편하게 적용할 수 있다는 점도 실용적이다.

약점은 데스크톱 GPU 지원이 상대적으로 약하고, LLM보다 컴퓨터 비전 태스크에 더 최적화돼 있다는 점이다.

2.2.3 ONNX Runtime

ONNX Runtime은 마이크로소프트가 주도하는 범용 추론 엔진이다. ONNX 형식의 모델이라면 다양한 하드웨어에서 일관된 성능을 낸다. CPU, CUDA, Core ML, NNAPI 등 백엔드를 자동으로 고른다.

강점은 **크로스 플랫폼 일관성**이다. Windows, Linux, macOS, iOS, Android 모두에서 동일한 모델을 동작시킨다. 프로파일링 도구가 우수해 성능 병목을 쉽게 파악할 수 있다. 디바이스 자동 선택 기능이 있어 별도의

하드웨어 설정 없이도 최적의 백엔드를 골라 쓴다.

약점은 LLM 최적화보다 전통적인 ML 모델에 초점이 더 맞춰져 있다는 점이다. 최신 LLM 아키텍처 지원이 Llama.cpp보다 느린 경우가 있다.

2.2.4 TensorFlow Lite

TensorFlow Lite는 텐서플로우의 모바일/임베디드용 포크다. 주로 컴퓨터 비전 모델에 강점을 보인다. 안드로이드 NNAPI와 iOS Core ML을 모두 지원한다.

강점은 텐서플로우 에코시스템과의 긴밀한 통합이다. 텐서플로우로 학습한 모델을 쉽게 변환해 배포할 수 있다. 안드로이드 NNAPI를 이용하면 Hexagon DSP, GPU, NPU 등 다양한 하드웨어 가속을 활용할 수 있다.

약점은 LLM 추론에는 그다지 최적화돼 있지 않다는 점이다. 변환 과정에서 일부 연산자가 지원되지 않아 모델을 수정해야 하는 경우도 있다.

2.2.5 Apple Core ML

Core ML은 iOS/macOS/watchOS/tvOS 전용 추론 프레임워크다. Metal Performance Shaders와 연계해 GPU 가속을 제공한다. Neural Engine이 탑재된 실리콘 맥에서는 특히 뛰어난 에너지 효율로 추론을 수행한다.

강점은 Apple 디바이스에서의 **에너지 효율성**이다. Neural Engine은 적은 전력으로 고효율 추론을 하므로 배터리 수명에 미치는 영향이 적다. Swift와 Objective-C에 쉽게 통합되며 Xcode와도 매끄럽게 연동된다.

약점은 Apple 플랫폼 전용이라는 점이다. Android나 Windows에서는 사용할 수 없으므로 크로스 플랫폼 앱이라면 별도의 엔진 선택이 필요하다.

2.3 양자화 레벨별 성능 비교

양자화는 모델 크기와 메모리 사용량을 줄이면서 추론 속도를 높이는 핵심 기법이다. 그러나 양자화 수준이 너무 높으면 생성 품질이 저하된다. 그래서 적절한 양자화 수준을 고르는 일이 중요하다.

양자화	크기 축소	품질 유지	속도 향상
FP16	~50%	거의 100%	1.2~1.5x
INT8	~75%	95~98%	1.5~2x
INT4	~87%	85~92%	2~3x
GPTQ	~87%	90~95%	2~4x
AWQ	~87%	93~97%	2~4x

FP16은 반정밀도 부동소수점 연산이다. 대부분의 GPU에서 하드웨어 지원을 받으므로 큰 속도 저하 없이 모델 크기를 절반으로 줄일 수 있다. 품질 저하가 거의 없다는 것이 강점이다.

INT8은 8비트 정수 연산이다. 전용 가속 하드웨어가 있는 디바이스에서는 속도가 눈에 띄게 빨라진다. 캘리브레이션 데이터셋으로 양자화 파라미터를 정하는데, 이 과정에서 품질이 약간 떨어질 수 있다.

INT4는 4비트 정수 연산이라 훨씬 더 극단적으로 압축할 수 있다. 그러나 양자화 에러가 누적돼 생성 품질이 눈에 띄게 떨어질 수 있다. 메모리 제약이 큰 환경에서 마지막 수단으로 고려한다.

GPTQ와 AWQ는 양자화 후 학습(Fine-tuning after Quantization) 기법이다. 양자화 과정에서 발생하는 품질 저하를 최소화하면서 INT4 수준의 크기 축소를 달성한다. 특히 AWQ는 Activation-Aware Weight Quantization으로 더 나은 품질을 보인다.

2.4 Apple Silicon Neural Engine 활용 전략

애플 실리콘 기반 Mac에서는 세 가지 추론 경로를 활용할 수 있다.

CPU 추론은 범용 코어에서 동작하므로 유연하지만 가장 느리다. 소규모 모델이나 디버깅 시 적합하다.

GPU 추론은 Metal Performance Shaders를 활용한다. Metal은 애플의 저수준 그래픽 API로, GPU 코어를 최대한 활용할 수 있다. Llama.cpp의 Metal 백엔드를 활성화하면 Mac GPU로 추론 속도를 크게 높일 수 있다.

Neural Engine 추론은 애플 실리콘에 탑재된 NE를 활용한다. 소모 전력이 적고 효율적이지만, 지원하는 연산에 제약이 있다. 특히 Core ML 모델만 NE를 활용할 수 있다.

최적의 전략은 **모델을 Core ML 형식으로 변환**하면서 NE에 최적화된 연산자로 구성하는 것이다. 그러면 NE의 에너지 효율성과 GPU의 병렬 처리 능력을 함께 활용할 수 있다.

2.5 Android NNAPI와 Qualcomm Hexagon 기반 가속

안드로이드에서 로컬 AI 추론을 가속하는 주된 경로는 NNAPI다. NNAPI는 안드로이드의 신경망 추론 추상화 레이어로, 디바이스의 가속 하드웨어를 일관된 API로 활용할 수 있게 한다.

퀄컴 스냅드래곤 시리즈에는 Hexagon DSP가 탑재되어 있다. NNAPI 델리게이션으로 Hexagon에서 추론을 실행하면 CPU보다 속도와 에너지 효율이 눈에 띄게 좋아진다.

구체적인 활용 단계는 다음과 같다. 먼저 TensorFlow Lite나 ONNX 모델을 .tflite나 .onnx 형식으로 변환한다. 이때 NNAPI 호환 모드로 내보낸다.

런타임에서 NNAPI Delegate를 활성화하면 디바이스가 Hexagon, GPU, CPU 중 최적의 가속 경로를 자동으로 선택한다.

모든 안드로이드 디바이스가 NNAPI를 잘 지원하는 것은 아니다. 특히 구형 디바이스에서는 NNAPI 지원이 제한적이거나 성능이 불안정할 수 있다. 따라서 런타임에서 NNAPI 가속 시도 후 실패 시 CPU 폴백 경로를 구현하는 것이 안전하다.

2.6 엔진 선택 결정 트리

프로젝트에 맞는 엔진은 요구사항에 따라 달라진다.

iOS만 지원한다면 Core ML이 우선 후보다. 에너지 효율성과 긴밀한 Xcode 통합이 강점이다.

Android만 지원한다면 NNAPI 가속이 되는 TensorFlow Lite나 ONNX Runtime을 고려한다. Qualcomm 디바이스가 목표라면 MNN도 좋다.

크로스 플랫폼이 필요하면서 LLM 추론이 목적이라면 Llama.cpp가 가장 확실한 선택이다. 활발한 커뮤니티와 다양한 모델 지원이 장점이다.

범용 ML보다 컴퓨터 비전 태스크가 주력이라면 TensorFlow Lite나 ONNX Runtime 쪽 생태계가 더 넓다.

3 모델 선택과 최적화 파이프라인

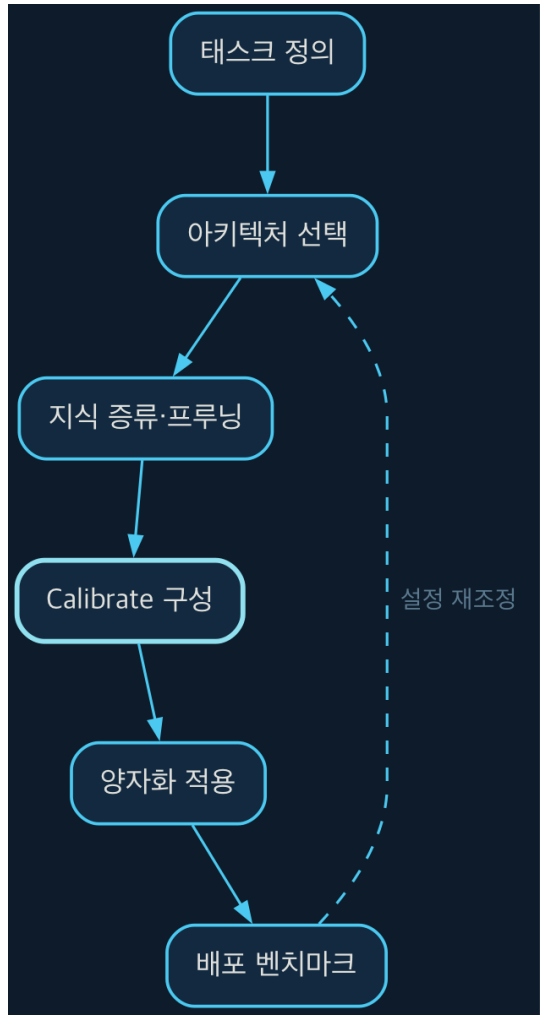

Figure 3.1: 모델 선택과 최적화 파이프라인

3.1 태스크별 최적 모델 아키텍처

로컬 AI에서 모델 선택은 성능을 결정하는 가장 근본적인 요소다. 먼저 자신의 앱에서 수행하려는 태스크를 명확히 정의해야 한다. 텍스트 생성이냐, 텍스트 분류냐, 이미지 인식이냐, 음성 처리이냐에 따라 적합한 모델이 완전히 달라진다.

텍스트 생성이 목적이라면 Decoder-only 트랜스포머 기반의 autoregressive 모델이 적합하다. Llama, Mistral, Qwen 계열이 대표적인 선택지다. 모델 크기는 디바이스 메모리에 따라 결정하는데, 스마트폰에서는 7B 파라미터 이하, 데스크톱에서는 13B 정도까지가 현실적인 범위다.

텍스트 분류나 감정 분석처럼 생성보다는 이해가 목적이라면 BERT 계열의 인코더 모델이 더 효율적이다. DistilBERT나 TinyBERT 같은 경량 모델은 모바일에서도 충분히 빠른 추론이 가능하다. 질문-답변 태스크에는 RoBERTa 계열도 좋은 선택이다.

이미지 인식은 ViT(Vision Transformer)나 CNN 기반 모델 중에서 선택한다. MobileViT나 EfficientNet은 모바일에 최적화된 경량 비전 모델로, 일반적인 이미지 분류나 객체 탐지에 적합하다. 실시간성이 요구되는 용도에는 YOLO 같은 one-stage detector가 유리하다.

음성 처리는 Whisper 계열이 텍스트 변환에서 강세를 보인다. 경량 버전인 tiny.en이나 base는 로컬에서도 충분히 실용적인 속도를 보인다.

3.2 지식 증류와 프루닝

큰 모델을 작은 디바이스에 맞게 압축하는 방법 중 하나가 지식 증류다. 대규모 teacher 모델이 가르치는 지식을 경량 student 모델에 전달하는 과정이다.

Self-Distillation은 teacher와 student가 같은 아키텍처를 쓰되 teacher의

soft label을 학습에 활용하는 방식이다. Temperature를 높여 확률 분포를 부드럽게 만들면 student가 그 분포를 학습하면서 teacher가 가진 암묵지까지 넘겨받는다.

Task Distillation은 태스크에 특화된 지식을 전달하는 방식이다. 기계번역 태스크라면 큰 모델이 생성한 번역 쌍을 데이터로 삼아 작은 모델을 학습시키는데, teacher가 만든 데이터가 student 모델 학습에 곧바로 쓰이는 셈이다.

Pruning은 모델에서 중요도가 낮은 가중치를 제거하는 기법이다. magnitude pruning은 작은 가중치 값을 걷어내는 방식이고, structured pruning은 필터나 헤드 단위로 통째로 제거해 하드웨어 친화적인 모델을 만든다. 반면 비structured pruning은 압축률은 더 높지만 하드웨어 가속이 어려워 오히려 속도가 느려질 수 있다.

실무에서는 pruning 이후 재학습(fine-tuning)으로 제거된 capacity를 보충하는 경우가 많다. 증류와 병행하면 품질 손실을 최소화하면서 크기를 줄일 수 있다.

3.3 양자화 실전

양자화 실전에서 가장 중요한 단계는 Calibrate 데이터셋 구성이다. Calibrate 데이터는 양자화 파라미터를 결정하는 데 사용되는 대표적인 입력 집합이다. 모델이 실제로 보는 입력 분포를 반영해야 양자화 에러가 최소화된다.

예를 들어 챗봇 앱이라면 실제 사용자 프롬프트의 분포를 반영한 데이터셋을 구성해야 한다. 무작위로 생성한 프롬프트보다는 실제 운영에서 수집한 프롬프트 중 대표성 있는 샘플을 추출하는 것이 좋다.

INT8 양자화의 구체적인 단계는 다음과 같다. 먼저 대표 입력 데이터 100~1000개를 수집하고, 이 데이터를 모델에 통과시키며 각 레이어의

activation 분포를 기록한다. 기록된 분포를 바탕으로 양자화 스케일과 Zero-point를 정하고, 그 파라미터로 양자화된 모델을 만든다. 마지막으로 원본 모델과 양자화 모델의 출력을 비교해 품질 저하를 확인한다.

Calibration 데이터가 실제 입력과 다르면 양자화 에러가 커진다. 예를 들어 Calibrate 데이터가 영어로만 구성되어 있는데 실제 입력에 한국어가 포함되어 있다면, 한국어 입력에서 양자화 품질이 급격히 저하될 수 있다. 다국어 앱이라면 언어별로 균등한 Calibrate 데이터를 구성해야 한다.

3.4 모바일 배포 최적화 체크리스트

모델을 모바일에 배포하기 전에 다음 항목을 점검해야 한다.

모델 크기는 디바이스 사용 가능 메모리보다 충분히 작아야 한다. OS와 다른 앱이 사용하는 메모리를 고려하면 실제 사용 가능한 메모리는 디바이스 총 메모리의 50~70% 수준이다. 메모리 할당 과정에서는 fragmentation도 발생하므로 여유분을 넉넉히 둔다.

메모리 풋프린트는 추론 중 피크 메모리 사용량을 의미한다. 모델 크기보다 중요한 지표로, kv cache 크기나 중간 activation 크기가 이를 좌우한다. 작은 배치 크기나 슬라이딩 윈도우 attention으로 메모리 사용량을 줄일 수 있다.

추론 지연 목표를 명확히 설정한다. 실시간 음성 처리라면 300ms 이하, 챗봇이라면 2초 이하 등 태스크별로 목표가 다르다. 벤치마킹을 통해 목표를 달성할 수 있는지 먼저 확인한다.

양자화 수준별 추론 속도를 측정한다. FP16으로 충분한 속도가 난다면 INT8이나 INT4까지 내려갈 필요가 없다. 양자화 레벨이 높을수록 크기는 줄지만 품질 저하와 속도 향상 사이의 trade-off가 존재한다.

최적의 설정은 디바이스별로 다를 수 있다. 그래서 다양한 설정으로

벤치마크를 수행한 뒤, 디바이스 하드웨어에 맞춰 최적 설정을 고르는 adaptive한 접근이 바람직하다.

4 하이브리드 아키텍처: 온라인과 오프라인을 넘나드는 설계

Figure 4.1: 하이브리드 아키텍처: 온라인과 오프라인을 넘나드는 설계

4.1 하이브리드 접근의 필요성

로컬 AI만으로는 부족할 때가 많다. 최신 지식에 기반한 답변이 필요하거나, 고사양 GPU 없이는 처리하기 힘든 복잡한 추론이 필요한 경우가 그렇다. 이럴 때 클라우드 AI를 함께 쓰면 두 방식의 장점을 모두 살릴 수 있다.

하이브리드 아키텍처의 핵심은 **언제 로컬을 쓰고 언제 클라우드를 쓰는지**를 명확히 정의하는 데 있다. 이 분기 기준이 모호하면 런타임에 예기치 않은 동작이 벌어진다. 전환 과정에서 사용자 경험이 끊기지 않도록 설계하는 것도 중요하다.

4.2 폴백 전략 설계

폴백 전략은 클라우드 서비스를 쓸 수 없을 때 로컬 추론으로 서비스를 계속 유지하는 메커니즘이다. 이를 설계할 때는 세 가지 시나리오를 염두에 둬야 한다.

첫 번째는 네트워크가 완전히 끊기는 상황이다. 비행기 안이거나 엄격한 네트워크 격리가 요구되는 환경에서는 클라우드에 접속할 수 없다. 이때는 로컬 추론이 유일한 대안이다. 다만 로컬 모델의 성능은 제한적이라 모든 기능이 그대로 동작하지는 않는다. 그래서 **핵심 기능만 로컬에서 제공하고, 고급 기능은 비활성화**하는 편이 낫다.

두 번째는 네트워크 지연이 과도한 상황이다. LTE처럼 불안정하거나 지연이 큰 환경에서는 클라우드 응답이 느려진다. 타임아웃 임계값을 정해두고 일정 시간 안에 응답이 없으면 로컬 추론으로 넘어가면 사용자 경험이 한결 나아진다.

세 번째는 클라우드 서비스가 일시적으로 장애를 일으키는 경우다. API 서비스가 다운되거나 Rate Limit에 걸렸을 때가 여기에 해당한다. 이때 로컬 폴백이 즉시 동작하면 서비스 중단을 최소화할 수 있다.

구현에서는 AsyncTimeout 패턴을 쓴다. 클라우드 요청을 비차단 방식으로 보내고 설정된 시간 안에 응답이 없으면 로컬 추론 결과를 반환한다. 클라우드 응답이 뒤늦게 도착하면 무시하거나 캐시에 저장해 다음에 활용하면 된다.

4.3 스마트 캐싱

자주 반복되는 쿼리는 로컬 추론 결과를 캐싱해두면 응답 속도를 크게 높일 수 있다. 클라우드 비용도 함께 줄어든다. 캐시 설계에서 핵심은 **무효화 정책**과 **저장 용량 관리**다.

가장 간단한 캐시 구조는 LRU(Least Recently Used)다. 최근에 사용된 항목을 유지하다가 용량이 넘치면 오래된 항목부터 지운다. 구현이 간단하면서도 효과적이다.

더 고도화된 캐싱 전략으로는 의미론적 캐싱이 있다. 사용자 쿼리의 의미적 유사도를 계산해 비슷한 쿼리의 기존 결과를 재활용하는 방식이다. 벡터 데이터베이스나 근접 이웃 검색으로 구현한다. 쿼리가 완전히 같지 않아도 의미가 비슷하면 캐시 히트로 처리할 수 있다.

계층적 캐시 구조를 추천한다. 메모리에 두는 L1 캐시, 디스크의 L2 캐시, 원격 캐시 서버 세 단계로 구성한다. 빠른 응답이 필요하면 메모리 캐시부터 확인하고, 미스가 나면 디스크, 그래도 없으면 원격을 확인하는 순서다.

4.4 데이터 선처리

오프라인 상태에서도 AI 기능이 원활하게 동작하려면 미리 처리해둘 수 있는 작업을 파악해 선처리해두어야 한다.

가장 효과적인 선처리 대상은 **사용자가 반복적으로 요청할 가능성이 높은 콘텐츠**다. 예를 들어 챗봇 앱이라면 자주 묻는 질문(FAQ)의 응답을 미리

생성해 캐싱해두면 좋다.

콘텐츠 인덱싱도 미리 해둘 수 있다. 문서 요약이나 검색 색인 구축처럼 시간이 오래 걸리는 작업을 네트워크가 연결됐을 때 미리 끝내두면 오프라인 상태에서도 응답이 빠르다.

Predictive preprocessing은 사용자의 다음 행동을 예측해 미리 처리해두는 고급 기법이다. 앱 사용 패턴을 분석해 다음에 필요할 법한 데이터를 미리 내려받아 처리한다. 예를 들어 사용자가 글을 자주 읽는 시간대에 맞춰 관련 콘텐츠를 미리 로드해두는 식이다.

4.5 사용자 경험 일관성 확보

온라인과 오프라인을 오갈 때 사용자가 불필요하게 다시 입력하지 않도록 설계해야 한다. 상태 관리와 입력 보존 전략이 이를 뒷받침한다.

입력 상태는 항상 로컬에 보관한다. 사용자가 작성 중인 메시지나 업로드를 기다리는 이미지는 디바이스에 저장해둔다. 클라우드 연결이 복구되면 자동으로 동기화해 사용자가 따로 다시 입력할 필요가 없게 한다.

응답의 출처를 시각적으로 표시한다. 로컬 추론 결과인지 클라우드 API에서 가져온 결과인지 구분할 수 있으면, 사용자가 이 응답을 어느 정도 신뢰할지 판단하는 데 도움이 된다.

대규모 태스크는 나눠서 처리한다. 완전한 오프라인 동작이 어렵다면 부분적으로라도 기능을 유지하는 편이 낫다. 예를 들어 문서 요약 앱에서 클라우드 연동이 끊기면 짧은 문서만 로컬에서 처리하고, 긴 문서는 사용자에게 오프라인 상태를 알린 뒤 클라우드가 복구되면 처리 일정을 잡아준다.

4.6 하이브리드 설계 패턴 요약

성공적인 하이브리드 설계는 다음 원리를 따른다.

로컬 우선, 클라우드 보완이 기본 철학이다. 가능한 한 많은 기능을 로컬에서 처리하고, 로컬이 한계에 부딪혔을 때만 클라우드로 확장한다.

실패는 graceful하게 처리한다. 클라우드 호출이 실패해도 앱 전체가 crash하지 않고 로컬 결과나 에러 메시지를 반환한다.

사용자 oblivious하게 동작한다. 온라인과 오프라인 사이를 사용자가 의식하지 않아도 되도록 자동으로 넘나들되, 필요할 때는 상태를 눈에 보이게 해준다.

5 구체적 구현 사례: Python과 Swift/Android 코드 예제

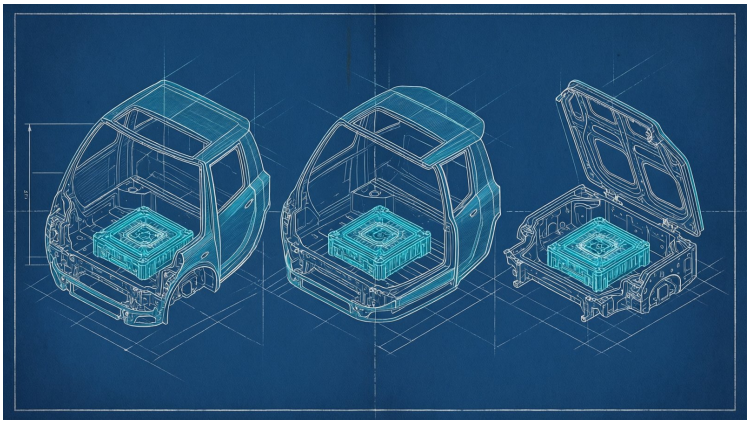

Figure 5.1: 구체적 구현 사례: Python과 Swift/Android 코드 예제

5.1 Llama.cpp Python 바인딩으로 간단한 챗봇 구현

Llama.cpp의 Python 바인딩을 활용하면 간단하게 로컬 챗봇을 구현할 수 있다. 먼저 llama-cpp-python 패키지를 설치한다.

```
pip install llama-cpp-python
```

GGUF 형식의 모델 파일을 내려받아 구성한다. TheBloke의 Hugging Face에서 다양한 양자화 모델을 구할 수 있다.

기본적인 챗봇 구현 예시는 다음과 같다.

```
from llama_cpp import Llama
```

```
def __init__(self, model_path: str):
 self.llm = Llama(
 model_path=model_path,
 n_ctx=2048,
 n_threads=4,
 use_mmap=True,
 use_mlock=False,
 )

def generate(self, prompt: str, max_tokens: int = 256) -> str:
 output = self.llm(
 prompt,
 max_tokens=max_tokens,
 stop=["</s>", "USER:", "SYSTEM:"],
 echo=False,
 )
 return output["choices"][0]["text"].strip()
```

초기화 단계에서 모델 경로와 컨텍스트 크기, 스레드 수를 설정한다. use_mmap는 메모리 매핑을 활성화하여 큰 모델도 메모리에 전부 올리지 않고 부분적으로 로드할 수 있게 한다.

실제 제품이라면 세션 관리와 프롬프트 템플릿을 추가해야 한다. 시스템 프롬프트로 역할을 정의하고, 대화 이력을 컨텍스트에 포함시키는 구조가 일반적이다.

5.2 iOS Core ML 통합

iOS에서 Core ML 모델을 활용하려면 먼저 모델을 .mlmodel 형식으로 변환해야 하는데, coremltools를 사용하면 된다.

```
import coremltools as ct

traced_model = torch.jit.trace(model, example_input)
mlmodel = ct.convert(
 traced_model,
 inputs=[ct.TensorType(shape=example_input.shape)],
)
mlmodel.save("MyModel.mlpackage")
```

변환된 .mlpackage를 Xcode 프로젝트에 추가하면 Xcode가 자동으로 모델 리소스를 번들한다.

Swift에서 모델을 로드하고 추론하는 코드는 다음과 같다.

```
import CoreML
import NaturalLanguage

func predict(input: String) async throws -> String {
 let config = MLModelConfiguration()
 config.computeUnits =.all // CPU + GPU + Neural Engine
 let model = try NLModel(contentsOf: modelURL, configuration:
 ↪  config)

 let sentiment = model.predictedLabel(for: input)
 return sentiment ?? "neutral"
}
```

NLModel은 NaturalLanguage 프레임워크가 제공하는 추상화로, 텍스트 분류·감정 분석·언어 식별 같은 태스크에 쓸 수 있다. 커스텀 모델은 MLModel 클래스로 직접 로드해 추론하면 된다.

성능을 최대화하려면 computeUnits 설정이 중요하다. `.cpuAndNeuralEngine`로 설정하면 Neural Engine에서 추론이 실행되어 에너지

효율이 좋다. `.all`로 설정하면 GPU와 Neural Engine을 모두 활용하여 속도를 높인다.

5.3 Android NNAPI 실전

Android에서 TensorFlow Lite 모델을 NNAPI 가속으로 실행하는 예시를 본다. 먼저 build.gradle에 의존성을 추가한다.

```
implementation 'org.tensorflow:tensorflow-lite:2.14.0'
implementation 'org.tensorflow:tensorflow-lite-support:0.4.4'
```

NNAPI Delegate를 활성화하여 추론을 실행한다.

```
import org.tensorflow.lite.Interpreter
import org.tensorflow.lite.nnapi.NnApiDelegate

class LocalInference(private val modelPath: String) {

 private val interpreter: Interpreter

 init {
 val options = Interpreter.Options()

 // NNAPI Delegate 추가
 val nnapiDelegate = NnApiDelegate.Builder()
.setPrecisionLossAllowed(false)
.setPowerPreference(PowerPreference.HIGH_PERFORMANCE)
.build()
 options.addDelegate(nnapiDelegate)

 // CPU 폴백 Delegate 추가
 options.setNumThreads(4)
```

```
 interpreter = Interpreter(File(modelPath), options)
 }

 fun run(input: FloatArray): FloatArray {
 val output = FloatArray(outputSize)
 interpreter.run(input, output)
 return output
 }
}
```

NNAPI Delegate가 동작하지 않는 구형 기기에서는 CPU 추론으로 자동 폴백된다. `setPrecisionLossAllowed(false)`로 FP32 정밀도 유지를 요청하지만, NNAPI가 INT8 연산만 지원하면 자동으로 양자화되어 실행된다.

5.4 성능 벤치마킹 프레임워크 구축

자신의 디바이스에서 모델 성능을 정확히 측정하려면 체계적인 벤치마킹 프레임워크가 필요하다. 벤치마크는 추론 지연(latency)과 처리량(throughput)을 모두 측정해야 한다.

```
import time
import statistics

class ModelBenchmark:
 def __init__(self, model, warmup_runs=5, measure_runs=20):
 self.model = model
 self.warmup_runs = warmup_runs
 self.measure_runs = measure_runs
```

```
def measure_latency(self, input_data):
# 워밍업
for _ in range(self.warmup_runs):
self.model.generate(input_data)

# 실제 측정
latencies = []
for _ in range(self.measure_runs):
start = time.perf_counter()
self.model.generate(input_data)
end = time.perf_counter()
latencies.append((end - start) * 1000) # ms 단위

return {
"mean_ms": statistics.mean(latencies),
"median_ms": statistics.median(latencies),
"p95_ms": sorted(latencies)[int(len(latencies) * 0.95)],
"min_ms": min(latencies),
"max_ms": max(latencies),
}

def measure_throughput(self, input_data, total_tokens):
start = time.perf_counter()
tokens_generated = 0
while tokens_generated < total_tokens:
output = self.model.generate(input_data)
tokens_generated += len(output.split())
end = time.perf_counter()

elapsed = end - start
return {
"tokens_per_second": total_tokens / elapsed,
```

```
"total_time_s": elapsed,
}
```

벤치마크 결과를 기록하여 디바이스 간, 모델 간 성능을 비교한다. 특히 모바일에서는 thermal throttling이 발생할 수 있으므로, 연속 실행보다는 간헐적으로 측정해 열적 영향을 피하는 편이 안전하다.

5.5 마무리: 통합 아키텍처 예시

로컬 AI 기능을 앱에 통합하는 전체 파이프라인을 정리하면 다음과 같다.

먼저 모델을 선정한다. 앱의 핵심 태스크에 맞는 모델을 하드웨어 제약 안에서 고르는데, 7B 파라미터 이하의 INT4 양자화 모델이 스마트폰에서는 현실적인 출발점이 된다.

다음은 추론 엔진을 정하는 일이다. iOS라면 Core ML, Android라면 TF Lite와 NNAPI 조합, 크로스 플랫폼을 노린다면 Llama.cpp가 일반적으로 선택된다.

이어서 최적화 단계가 온다. 양자화와 프루닝으로 모델 크기와 메모리 사용량을 줄이고, Calibrate 데이터셋을 구성해 양자화 품질을 확인한다.

그다음은 하이브리드 설계다. 클라우드 폴백 경로를 마련하고 스마트 캐싱으로 응답 속도를 높이며, 오프라인 우선 동작을 보장해야 한다.

마지막은 벤치마크다. 실제 디바이스에서 지연과 처리량을 측정하고 다양한 설정을 실험하며 최적 구성을 찾아간다.

로컬 AI 개발은 클라우드 개발과는 다른 종류의 도전을 마주한다. 하드웨어 제약, 플랫폼별 최적화, 다양한 디바이스 환경 대응이 요구된다. 하지만 이 도전을 넘어서면 데이터 프라이버시와 응답 속도, 서버 비용 절감이라는 확실한 가치를 얻는다. 이 책이 독자의 로컬 AI 애플리케이션 개발에

실질적인 도움이 되기를 바란다.