5장 제미나이 Function Calling (함수호출)

거대언어모델(LLM)에서 제공하는 함수 호출(Function Calling) 기능은 인공지능이 API를 통해 외부 세계와 쉽게 연동할 수 있도록 도와줍니다. 외부 세계란 시스템 내부의 리소스, 인터넷을 통해 접근하는 정보, 사물을 작동시키는 신호 등을 포함합니다.

5.1 함수호출 기초

다음처럼 구글 제미나이 API에서 제공하는 Function Calling 자동 호출 파라미터를 활용하면 함수 호출을 손쉽게 사용할 수 있습니다.

```python
import google.generativeai as genai

genai.configure(api_key="[redacted]")

def get_price(product: str)-> int:
    """제품의 가격을 알려주는 함수

    Args:
        theme: 제품명
    """
    return 1000


def get_temperature(city: str)-> float:
    """도시의 온도를 알려주는 함수

    Args:
        genre: 도시명
    """
    return 20.5

model = genai.GenerativeModel(model_name="gemini-1.5-flash", tools=[get_price, get_temperature])
chat_session = model.start_chat(enable_automatic_function_calling=True)
response = chat_session.send_message("서울의 온도는?")
print(response.text)
```

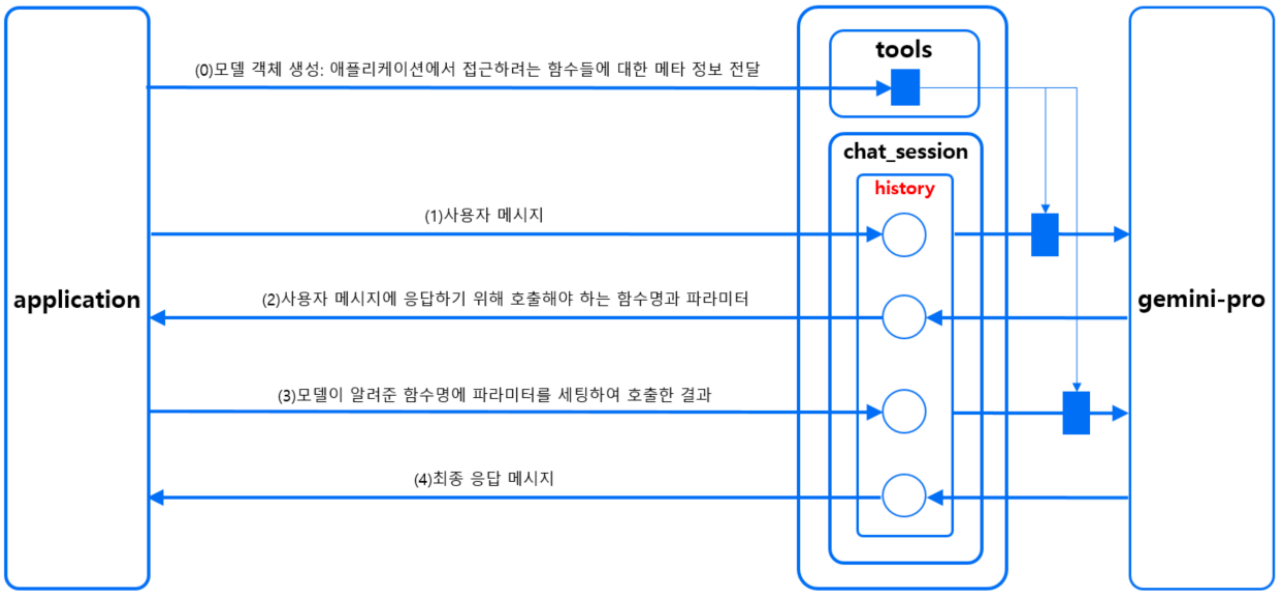
```
서울의 온도는 20.5도입니다.
```

docstring을 사용해 함수의 기능과 매개변수를 기술하고 tools에 함수를 파라미터로 전달한 다음, 서울의 온도를 물으면 제미나이는 get_temperature라는 메서드의 결과를 바탕으로 답변을 생성합니다. 내부적으로는 doctring 외에도 함수 이름, 매개변수, 반환 타입까지 고려하니, 적확한 함수명은 물론 타입 어노테이션을 꼼꼼히 기술해야 합니다. 이와 같이 enable_automatic_function_calling=True로 수행하면 간편하기는 하지만, 멀티턴 대화에서 잘 작동하지 않는 경우가 있고, 무엇보다도 개발자가 함수 호출 과정을 세밀하게 제어할 수 없다는 단점이 있습니다. 따라서 서비스에 함수 호출을 적용하고자 하면, 함수 호출의 작동 원리를 이해한 후 수동(enable_automatic_function_calling=False)으로 함수 호출을 사용하는 것이 좋습니다.

5.2. 거대언어모델의 함수 호출 과정

거대언어모델을 통해 함수를 호출하는 과정은 다음과 같습니다. 1. 프로그램에서 모델 객체 생성 시 접근하려는 함수들의 메타 정보(함수명, 파라미터, 설명)를 설정하고 전달합니다. 2. 사용자가

메시지를 전달할 때마다 메시지 내용과 함께 메타 정보가 모델에게 전달됩니다. 3. 모델은 사용자 메시지와 메타 정보를 분석해 함수 호출이 필요하다고 판단되면 호출할 함수명과 파라미터값을 반환합니다. 4. 프로그램은 모델이 지정한 대로 함수를 호출하고 그 결괏값을 다시 모델에게 전달합니다. 5. 모델은 함수 결괏값을 참조해 사용자에게 전달할 최종 메시지를 생성합니다. 참고로, 함수를 파라미터로 넘기면 SDK에서 내부적으로 각 함수의 메타 정보를 분석해 모델에 넘깁니다.



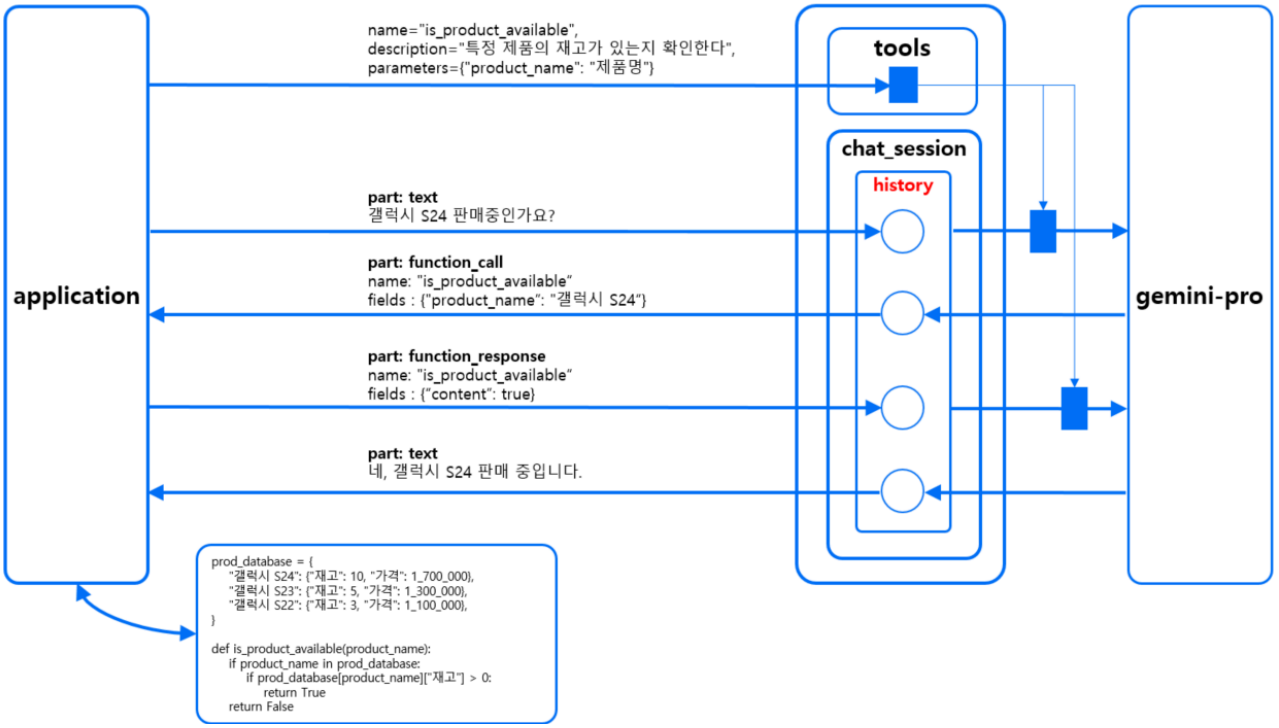
그림에서 볼 수 있듯이 이 모든 과정은 대화 이력(history)에 쌓이며, 모델을 호출할 때마다 대화이력과 함수의 메타 정보가 함께 전달되므로 토큰 사용량이 그만큼 더 늘어난다는 점은 알고 있는 것이 좋습니다.

5.2.1. 함수 호출 예시

갤럭시 S24 판매 질의에 대한 챗봇의 대응 과정을 예로 들어 봅시다.

사용자가 챗봇에 “갤럭시 S24 판매 중인가요?”라고 질문했다면 모델은 이 질문이 특정 제품의 재고를 묻는 것임을 인식하고, 정확한 답변을 위해서는 실시간 재고 정보가 필요하다고 판단합니다. 따라서 모델은 애플리케이션에게 데이터베이스에 접근하여 갤럭시 S24의 재고를 확인하는 함수를 호출하도록 요청합니다.

애플리케이션은 모델의 요청에 따라 데이터베이스에서 해당 제품의 재고를 조회하는 함수를 실행합니다. 함수 실행 결과로 얻은 재고 정보를 모델에 전달하면, 모델은 그 정보를 바탕으로 사용자에게 적절한 답변을 생성합니다. 예를 들면 “네, 갤럭시 S24 판매 중입니다.”와 같은 답변을 생성할 수 있습니다.




```
prod_database = {
    "갤럭시 S24": {"재고": 10, "가격": 1_700_000},
    "갤럭시 S23": {"재고": 5, "가격": 1_300_000},
    "갤럭시 S22": {"재고": 3, "가격": 1_100_000},
}

def is_product_available(product_name):
    if product_name in prod_database:
        if prod_database[product_name]["재고"] > 0:
            return True
    return False
```

5.3. 함수 호출 구현하기

앞의 다이어그램을 코드로 구현하겠습니다.

실제 함수 정의

데이터베이스와 함수 정의부입니다.

```python
import google.generativeai as genai
import google.ai.generativelanguage as glm

genai.configure(api_key="[redacted]")

prod_database = {
    "갤럭시 S24": {"재고": 10, "가격": 1_700_000},
    "갤럭시 S23": {"재고": 5, "가격": 1_300_000},
    "갤럭시 S22": {"재고": 3, "가격": 1_100_000},
}


def is_product_available(product_name: str)-> bool:
    """특정 제품의 재고가 있는지 확인한다.

    Args:
        product_name: 제품명
    """
    if product_name in prod_database:
        if prod_database[product_name]["재고"] > 0:
            return True
    return False

def get_product_price(product_name: str)-> int:
    """제품의 가격을 가져온다.

    Args:
        product_name: 제품명
    """
    if product_name in prod_database:
        return prod_database[product_name]["가격"]
    return None

def place_order(product_name: str, address: str)-> str:
    """제품 주문결과를 반환한다.
    Args:
        product_name: 제품명
        address: 배송지
    """
    if is_product_available(product_name):
        prod_database[product_name]["재고"] -= 1
        return "주문 완료"
    else:
        return "재고 부족으로 주문 불가"
```

다음은 모델이 JSON 문자열로 알려준 함수와 실제 함수를 연결하기 위한 매핑 정보입니다.

```
function_repoistory = {
    "is_product_available": is_product_available,
    "get_product_price": get_product_price,
    "place_order": place_order
}
```

모델 생성과 함수 전달

모델을 생성할 때 tools 필드에 함수 리스트를 세팅합니다.

```
model = genai.GenerativeModel(
    model_name="gemini-1.5-flash",
    tools=function_repoistory.values()
)
```

메시지 주고 받기

갤럭시 S24를 판매 중인지 질의하면, function_call 데이터 구조를 통해 호출할 함수명과 세팅할 파라미터 정보를 반환하는 예시입니다.

```
chat_session = model.start_chat()
prompt = "갤럭시 S23 판매 중인가요?"
response = chat_session.send_message(prompt)
print(response.candidates[0].content.parts[0])
# print(response.candidates)
```

결과화면

```
parts {
  function_call {
    name: "is_product_available"
    args {
      fields {
        key: "product_name"
        value {
          string_value: "갤럭시 S24"
        }
      }
    }
  }
}
role: "model"
```

응답으로 function_call 데이터 타입이 반환되었으므로 is_product_available(product_name="갤럭시 S24")와 같이 특정 함수가 호출되도록 해야 합니다. 이때 이러한 함수 호출 과정은 다음처럼 일반화하는 것이 재사용성을 높입니다.

```
part = response.candidates[0].content.parts[0]
print(part)
if part.function_call:
    function_call =  part.function_call
    function_name = function_call.name
    function_args = {k: v for k, v in function_call.args.items()}
    print(f"{function_name} args=>: {function_args}")
    function_result = function_repoistory[function_name](**function_args)
    print(f"{function_name} result=>: {function_result}")
```

respository에서 이름(Key)으로 함수(Value)를 가지고 온 후 딕셔너리 언패킹/패킹 테크닉을 활용하여 함수를 호출하고 있습니다. 그리고 이렇게 함수를 호출한 결과값은 다음과 같이 function_response 형태의 Part 데이터로 전달함으로써 모델이 결과값에 맞는 적절한 메시지를 생성할 수 있습니다.

```
part = glm.Part(
    function_response=glm.FunctionResponse(
        name="is_product_available",
        response={
            "content": function_result,
        },
    )
)
print(part)
response = chat_session.send_message(part)
# correct_response(response)
print("--------")
print(response.candidates[0].content)
```

결과화면

```
function_response {
  name: "is_product_available"
  response {
    fields {
      key: "content"
      value {
        bool_value: true
      }
    }
  }
}
-----
role: "model"
parts {
  text: "네, 갤럭시 S22는 판매 중입니다."
}
```

5.3.1 스마트폰 주문 챗봇 구현

지금까지 함수 호출이 한 번 일어날 때의 과정을 알아봤습니다. 그런데 사업적으로 의미 있는 챗봇을 구현하려면 멀티턴 대화 과정에서 여러 차례의 함수 호출이 적절히 일어나야 합니다. 이번 절에서는 여러 차례 함수 호출을 수행하는 과정에서 스마트폰 주문 처리 업무를 수행하는 챗봇을 구현하겠습니다.

실습에서 가정하는 스마트폰 주문을 위한 액티비티입니다.

① 특정 제품에 대한 재고 확인 ② 특정 제품에 대한 가격 확인 ③ 특정 제품에 대한 주문 신청

```
#-*-coding:utf-8 -*-
import google.generativeai as genai
import google.ai.generativelanguage as glm

genai.configure(api_key="AIzaSyD0jMkUWWxa6O1qpGjMIz80zOVxM4KoyKU")

prod_database = {
    "갤럭시 S26": {"재고": 10, "가격": 1_700_000},
    "갤럭시 S25": {"재고": 5, "가격": 1_300_000},
    "갤럭시 S24": {"재고": 3, "가격": 1_100_000},
}

def is_product_available(product_name: str)-> bool:
    """특정 제품의 재고가 있는지 확인한다.
    Args:
        product_name: 제품명
    """

    if product_name in prod_database:
        if prod_database[product_name]["재고"] > 0:
            return True
    return False

def get_product_price(product_name: str)-> int:
    """제품의 가격을 가져온다.

    Args:
        product_name: 제품명
    """
    if product_name in prod_database:
        return prod_database[product_name]["가격"]
    return None

def place_order(product_name: str, address: str)-> str:
    """제품 주문결과를 반환한다.
    Args:
        product_name: 제품명
        address: 배송지
    """
    if is_product_available(product_name):
```

```
        prod_database[product_name]["재고"] -= 1
        return "주문 완료"
    else:
        return "재고 부족으로 주문 불가"

function_repoistory = {
    "is_product_available": is_product_available,
    "get_product_price": get_product_price,
    "place_order": place_order
}

def correct_response(response):
    part = response.candidates[0].content.parts[0]
    if part.function_call:
        for k, v in part.function_call.args.items():
            byte_v = bytes(v, "utf-8").decode("unicode_escape")
            corrected_v = bytes(byte_v, "latin1").decode("utf-8")
            part.function_call.args.update({k:  corrected_v})

model = genai.GenerativeModel(
    model_name="gemini-1.5-flash",
    tools=function_repoistory.values()
)

chat_session = model.start_chat(history=[])
queries = ["갤럭시 S25 판매 중인가요?", "가격은 어떻게 되나요?", "경기 고양시 일산동구
중앙로1275번길 로 배송부탁드립니다"]

for query in queries:
    print(f"\n사용자: {query}")
    response = chat_session.send_message(query)
    correct_response(response)
    part = response.candidates[0].content.parts[0]

    if part.function_call:
        function_call =  part.function_call
        function_name = function_call.name
        function_args = {k: v for k, v in function_call.args.items()}
        function_result = function_repoistory[function_name](**function_args)
        part = glm.Part(
            function_response=glm.FunctionResponse(
                name=function_name,
                response={
                    "content": function_result,
                },
            )
        )
        response = chat_session.send_message(part)
        print(f"모델: {response.candidates[0].content.parts[0].text}")
```

앞서 설명했던 코드를 세 개의 비즈니스 요구에 세 개의 함수를 만들고 사용자 질의 시 해당하는함수가 호출되도록 구현했습니다. 함수 호출 응답 과정에서 한글이 유니코드 문자열로 대체되는 현상이 있어 correct_response라는 함수를 추가했습니다. 다음은 실행 결과입니다

사용자: 갤럭시 S24 판매 중인가요?
모델: 네, 갤럭시 S24가 판매 중입니다.

사용자: 가격은 어떻게 되나요?
모델: 갤럭시 S24의 가격은 1,700,000원 입니다.

사용자: 경기 고양시 일산동구 중앙로1275번길 로 배송부탁드립니다
모델: 감사합니다. 갤럭시 S24를 서울시 종로구 종로1가 1번지로 배송해드리겠습니다.
==========
history:

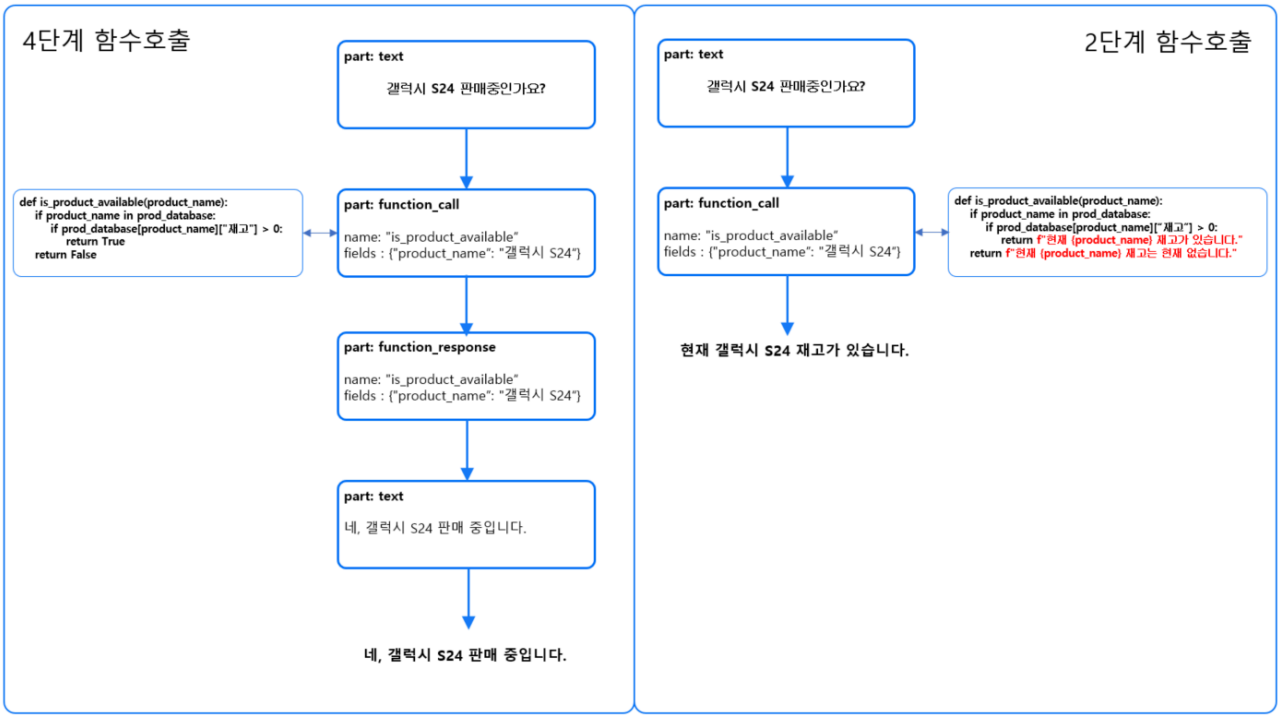
```
[
  role: "user"
  parts {
    text: "갤럭시 S24 판매 중인가요?"
  },
  role: "model"
  parts {
    function_call {
      name: "is_product_available"
      args {
        fields {
          key: "product_name"
          value {
            string_value: "갤럭시 S24"
          }
        }
      }
    }
  },
  role: "user"
  parts {
    function_response {
      name: "is_product_available"
      response {
        fields {
          key: "content"
          value {
            bool_value: true
          }
        }
      }
    }
  },
  role: "model"
  parts {
    text: "네, 갤럭시 S24가 판매 중입니다. \n"
  },
  ...이하 생략...
```

history를 보면 사용자(user)가 text 타입으로 질의를 하면, 모델(model)이 함수 호출 방법에 대해 funcion_call 타입으로 알려줍니다. 그러면 사용자는 function_response 타입으로 함수 호출

결과를 전달하고, 모델은 text 타입으로 최종 응답을 반환합니다. 앞서 다이어그램에서도 표현했지만, 이처럼 하나의 완결된 함수 호출은 text(user) → function_call(model) → function_response(user) → text(model)라는 4단계를 한 세트로 하여 구성됩니다.

5.3.2 2단계 함수 호출 구현하기

실무적으로 보면 사용자의 질문에 대답하는 데 필요한, 함수와 파라미터를 식별하는 2단계까지가 중요합니다. 대부분의 비즈니스용 챗봇은 화려한 언변을 뽐내는 것이 목적이 아니라 주어진 질문에 사실을 근거로 답하는 것이 중요하기 때문입니다. 가령 “갤럭시 S24 상품 있나요?”라는 질의가 들어 왔을 때 is_product_available(product_name=’갤럭시 S24’) 호출로 결괏값을 반환받았다면, 나머지 표현은 고정된 텍스트를 통해 출력하는 것이 더욱 안정적입니다. 뿐만 아니라 불필요한 모델 호출도 줄이는 이점이 있습니다.



하지만 2단계 함수 호출을 사용한다고 해서 text(user) → function_call → function_response → text(model)이라는 데이터 구조를 벗어날 수는 없습니다. 가령 function_call 응답을 받은 뒤 function_response 데이터 구조를 생략하고 모델과 text로 메시지를 주고받는다면 “400 Please ensure that function response turn comes immediately after a function call turn.”과 같은 오류가 발생합니다. 따라서 2단계 수행 후 3, 4단계의 데이터 구조를 임의로 생성하는 과정이 필요합니다.

2단계 함수 호출 구현하기

각 함수의 반환을 튜플 형식으로 바꿨습니다. 앞의 반환값은 3, 4단계 진행이 필요한지를 나타내며, 뒤의 반환값은 원래의 함수 반환 값입니다. 이 예제에서는 재고 여부에 대해 답하는 경우만 2단계로 호출하게 했습니다.

```
def is_product_available(product_name: str)-> bool:
    """특정 제품의 재고가 있는지 확인한다.
```

```
    Args:
        product_name: 제품명
    """
    if product_name in prod_database:
        if prod_database[product_name]["재고"] > 0:
            return False, f"현재 {product_name} 재고가 있습니다."
    return False, f"현재 {product_name} 재고는 현재 없습니다."

def get_product_price(product_name: str)-> int:
    """제품의 가격을 가져온다.

    Args:
        product_name: 제품명
    """
    if product_name in prod_database:
        return True, prod_database[product_name]["가격"]
    return True, None

def place_order(product_name: str, address: str)-> str:
    """제품 주문결과를 반환한다.
    Args:
        product_name: 제품명
        address: 배송지
    """
    if is_product_available(product_name):
        prod_database[product_name]["재고"] -= 1
        return True, "주문 완료"
    else:
        return True, "재고 부족으로 주문 불가"
```

다음은 3, 4단계를 위해 모델 호출이 필요한지 분기하는 로직을 적용하는 코드입니다. 만일 3, 4 단계가 필요 없는 경우 make_fc_history 함수를 호출해 3, 4단계에 해당하는 구조를 chat_session의 history 객체에 삽입되게 했습니다.

```
 if part.function_call:
        function_call =  part.function_call
        function_name = function_call.name
        function_args = {k: v for k, v in function_call.args.items()}
        is_required_3_4, function_result = function_repoistory[function_name](**function_args)
        part = glm.Part(
                function_response=glm.FunctionResponse(
                    name=function_name,
                    response={
                        "content": function_result,
                    },
                )
            )
        response = chat_session.send_message(part)
        if is_required_3_4:
            response = chat_session.send_message(part)
        else:
            response = make_fc_history(chat_session, part, function_result)
```

history에는 질의와 응답 모두 Content 타입의 객체가 들어가야 하므로 이에 대한 정보를 생성했습니다. 한편, “갤럭시 S24 재고가 있습니다”라는 프로그램에서 만들어낸 메시지 역시 제미나이 SDK의 응답 구조에 맞추면 이후 로직을 일관되게 가져갈 수 있는 이점이 있습니다. 따라서 사용자에게 전달될 메시지도 제미나이 SDK의 응답 형식에 맞추도록 구현했습니다.

```
def make_fc_history(chat_session, part, answer):
    content = glm.Content(parts=[part], role="user")
    chat_session.history.append(content)

    response = glm.GenerateContentResponse({
        "candidates": [{"content": {"role": "model", "parts": [{"text": answer}]}}]
    })
    chat_session.history.append(response.candidates[0].content)
    return response
```

다음은 실행 결과입니다.

```
사용자: 갤럭시 S24 판매 중인가요?
모델: 현재 갤럭시 S24 재고가 있습니다.

사용자: 가격은 어떻게 되나요?
모델: 170만원 입니다.

사용자: 경기 고양시 일산동구 중앙로1275번길 로 배송부탁드립니다
모델: 주문이 완료되었습니다.
```

임의로 만들어낸 데이터 구조가 포함된 첫 번째 함수 호출 과정에 대한 history 객체의 내용입니다.

```
[role: "user"
parts {
  text: "갤럭시 S24 판매 중인가요?"
}
, role: "model"
parts {
  function_call {
    name: "is_product_available"
    args {
      fields {
        key: "product_name"
        value {
          string_value: "갤럭시 S24"
        }
      }
    }
  }
}
, role: "user"
parts {
```

```
  function_response {
    name: "is_product_available"
    response {
      fields {
        key: "content"
        value {
          string_value: "현재 갤럭시 S24 재고가 있습니다."
        }
      }
    }
  }
}
, role: "model"
parts {
  text: "현재 갤럭시 S24 재고가 있습니다."
}
```

4단계 함수 호출과 동일하게 text(user) → function_call → function_response → text(model) 순서로 데이터가 들어 있는 것을 확인할 수 있습니다.