FastAPI for Machine Learning Introduction

Building and Deploying ML APIs

A Comprehensive Guide to FastAPI,
API Development, and ML Model Deployment

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Understanding APIs: Foundation Concepts

1.1 What Are APIs?

1.1.1 Formal Definition

API Definition

APIs are mechanisms that enable two software components (such as the frontend and backend of an application) to communicate with each other using a defined set of rules, protocols, and data formats.

1.1.2 Simple Explanation

In simple terms, think of an API as a **connector**:

- It's a connector that links two pieces of software
- It enables communication between different applications
- It follows specific rules and formats for data exchange

1.2 Common API Use Case: Websites

Let's understand APIs through a common example: websites.

1.2.1 Website Architecture Components

Every website has two main components:

1. **Frontend** (User-Facing)
    - What users interact with
    - Forms, buttons, videos
    - Comments section, search bars
    - Visual elements users can see and click
2. **Backend** (Behind-the-Scenes)
    - Business logic implementation
    - Database interactions
    - Search algorithms
    - Data processing
    - Core application functionality

1.2.2 How APIs Connect Frontend and Backend

Real-World Example: Udemy Course Search

Scenario: You visit Udemy and search for "AI Agents" courses

Step-by-Step Flow:

1. **User Action:**

- Go to Udemy website (Frontend)
- Type "AI Agents" in search bar
- Press Enter

2. **Request Journey:**
   - Your search request goes to the API
   - API picks up the request
   - API forwards request to Backend
3. **Backend Processing:**
   - Backend receives request from API
   - Searches database for "AI Agents" courses
   - Retrieves matching courses
   - Sends results back to API
4. **Response Journey:**
   - API receives results from Backend
   - API formats data (typically as JSON)
   - API sends formatted data to Frontend
5. **Display:**
   - Frontend receives formatted data
   - Displays course results to user

1.2.3 Key Observations

From the example above, notice:

- **Protocols**: HTTP protocol is used (since it's web-based)
- **Data Format**: JSON format for data exchange
- **Connector Role**: API acts as intermediary
- **Structured Communication**: Follows defined rules

1.3 The Restaurant Analogy

A perfect analogy to understand APIs:

Restaurant Analogy

The Setup:

Imagine you're at a restaurant:

- **You (Customer)** = Frontend
- **Kitchen (Chef)** = Backend
- **Waiter** = API

The Process:

1. **Customer Orders:**
   - You (Frontend) tell waiter (API) what you want
   - Waiter takes your order/request
2. **Waiter Communicates:**
   - Waiter (API) goes to kitchen (Backend)
   - Tells chef your order
3. **Kitchen Prepares:**
   - Chef (Backend) prepares food
   - Does the actual work
4. **Food Delivered:**
   - Chef gives food to waiter (API)
   - Waiter brings food to your table (Frontend)

Additional Elements:

- **Menu Card** = Protocol/Rules
  - What can be ordered
  - Prices, preparation time
  - Valid options
- **Plating Style** = Data Format
  - Food served in specific way
  - Consistent presentation
  - Like JSON formatting

1.4 Why Do APIs Exist?

Understanding the problem that APIs solve helps understand their importance.

Fundamental Principle

When learning anything new, always ask: **"Why does this exist?"**
Understanding the problem that necessitated a solution helps you appreciate and remember the solution better.

1.4.1 The Pre-API Era

Let's explore a specific example to understand why APIs were invented.

2 The Pre-API Era: Understanding the Problem

2.1 Case Study: IRCTC Railway Booking

Let's examine how applications were built before APIs existed, using IRCTC (Indian Railway Catering and Tourism Corporation) as our example.

2.1.1 The Initial System

System Requirements:

- IRCTC website (basic version)
- No booking functionality initially
- Simple feature: Find trains between two stations on a specific date

Purpose:

- User enters: Station A, Station B, Date
- System shows: All trains running between these stations on that date
- Very basic information retrieval

2.2 Building Without APIs: Monolithic Architecture

2.2.1 Step-by-Step Component Development

Step 1: Database

- Need a database to store all train information
- Contains: Stations, train schedules, routes
- Structure: "These are two stations, these are all trains between them"
- All information contained in this database

Step 2: Backend

- Need code to access and query the database
- Technology: Python, Java, or similar
- Core functionality: `fetch_trains()` function
- Function input: Two stations + date
- Function process: Searches database
- Function output: List of trains on that date

Backend Function Concept

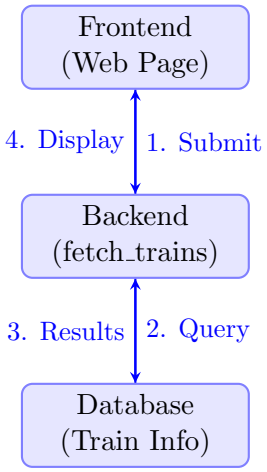
```
def fetch_trains(station1, station2, date):
    """
    Query database for trains between stations on given date
    Returns: List of trains
```

```
    """
    # Connect to database
    # Search for trains
    # Return results
    pass
```

Step 3: Frontend

- Web page for user interaction
- Contains a form with fields:
  - Field 1: First station name
  - Field 2: Second station name
  - Field 3: Date
  - Submit button
- Frontend connected to Backend
- When Submit pressed: HTTP request goes to Backend

2.2.2 The Complete Flow (Without API)



Process:

1. User submits form (Frontend)
2. HTTP request calls `fetch_trains()` function (Backend)
3. Function queries Database
4. Database returns information
5. Results sent back to Frontend
6. Frontend displays results to user

2.3 The Monolithic Architecture

What is Monolithic Architecture?

An architecture where:

- All code exists in a single folder/directory
- Frontend and Backend in same application
- Both developed within one project structure
- Tightly coupled components
- Everything bundled as one application

2.3.1 Key Characteristics

Single Application Structure:

- One folder/directory contains everything
- Frontend code in same location as Backend code
- No separation between components
- Deployed as single unit

Tightly Coupled:

- Frontend directly connected to Backend
- No intermediary layer needed
- Can communicate without API
- Changes in one affect the other

Important Note

Important Observation:

Notice that Backend and Frontend are communicating *without an API*. This is possible because they're tightly coupled within the same application. They're not separate software components.

2.3.2 Pre-API Website Development

Before APIs:

- All websites used Monolithic Architecture
- Frontend and Backend weren't separate software pieces
- They were parts of one unified software
- This architecture was called "Monolithic"

Warning

Technology Context:

For those familiar with PHP or Flask:

- If you've done web development with PHP (handling HTML files)
- Or used Flask to render HTML templates
- You've experienced Monolithic Architecture
- This is exactly what we're describing

2.4 The Problem with Monolithic Architecture

2.4.1 Initial Success

Good News:

- IRCTC website built and working
- Using Monolithic Architecture
- Website functioning properly
- Users can search for trains successfully

2.4.2 The Business Opportunity

New Development:

Several companies approach IRCTC:

- **MakeMyTrip**: Travel booking platform
- **Yatra**: Travel services company
- **Ixigo**: Travel search engine

Their Request:

- "Users ask us about train schedules too"
- "We don't have this information"
- "Only IRCTC has official railway data"
- "Can you give us access to this information?"
- "We'll pay per request"

IRCTC's Perspective:

- Great business opportunity
- Already providing data to our own users
- Other websites need it too
- Can earn money by sharing data
- Win-win situation

2.4.3 The Technical Challenge

The Problem:

How to technically implement this data sharing?

What Needs to Happen:

- MakeMyTrip, Yatra, and Ixigo need access to database information
- They are external software applications
- Need to access IRCTC's train schedule data

Option 1: Direct Database Access

Warning

Why This Won't Work:

- **Security Risk:** Cannot give direct database access to external companies
- **Data Integrity:** What if they accidentally modify data?
- **Control Loss:** No control over how they use the database
- **Not Feasible:** Too dangerous for production systems

Option 2: Backend Access

Remember our Backend has the `fetch_trains()` function:

- Give two stations and a date
- Returns trains between those stations
- Seems like a safer option

Ideal Scenario:

- MakeMyTrip software interacts with Backend component
- MakeMyTrip sends: Station names + Date
- Backend receives and processes request
- Backend queries Database
- Backend returns response to MakeMyTrip
- MakeMyTrip displays to their users

Same Process for Others:

- Yatra can do the same
- Ixigo can do the same
- Seems like the solution!

2.4.4 Why This Also Won't Work

Warning

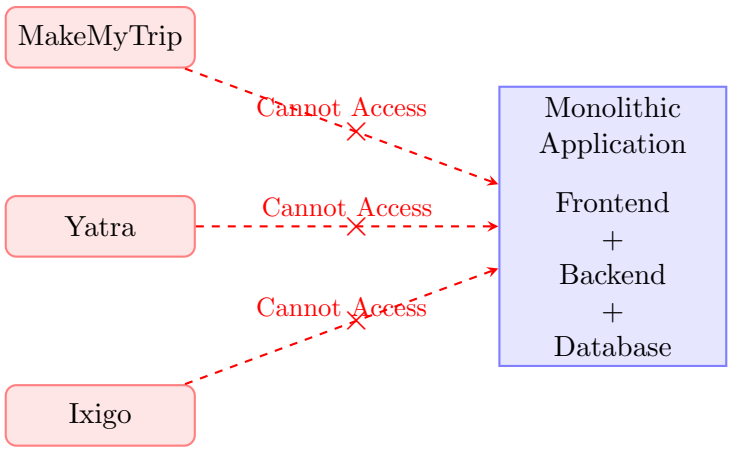
The Fundamental Problem:

The Backend is *not an independent application.* It's tightly coupled with the entire application.

Consequences:

- Backend is part of Monolithic application
- Backend is one small component of larger software
- Backend is tightly coupled with other components
- Cannot be accessed independently from outside
- External software cannot interact with it directly

2.5 The Dead End



Summary of the Problem:

- Cannot share Database directly (security risk)
- Cannot share Backend directly (tightly coupled)
- Cannot provide access to internal information
- External software cannot interact with monolithic system
- Information is locked inside the monolithic application

The Real Restriction

The information in the database is ONLY shareable within your own Monolithic application. External parties cannot access it. This is a huge business limitation.

Business Impact:

- IRCTC could earn money from other companies
- But the website architecture prevents it
- Massive revenue opportunity lost

- All because of technical limitations

Warning

This is precisely the problem that APIs solve.

The inability to share data and functionality with external applications is the exact issue that necessitated the invention of APIs.

3 The API Solution: Decoupled Architecture

3.1 Solving the Problem with APIs

Now let's see how APIs solve the problems we identified with Monolithic Architecture.

3.1.1 Step 1: Decoupling Backend from Frontend

The First Major Change:

Stop using Monolithic Architecture and decouple the application components.

Decoupling Means:

- Backend built separately
- Frontend built separately
- Backend = Different software application
- Frontend = Different software application
- No longer tightly coupled

New Architecture Components:

1. **Database**
   - Still exists as before
   - Contains all train information
   - Same data, same structure
2. **Backend (Now Independent)**
   - Separate software application
   - Contains `fetch_trains()` function
   - Can query database independently
   - Not connected to Frontend
   - Standalone application
3. **API Layer (NEW)**
   - Added in front of Backend
   - Makes Backend publicly accessible
   - Sits between Backend and Internet
   - Gateway to Backend functionality
4. **Frontend (Now Independent)**
   - Separate software application
   - No direct connection to Backend
   - Communicates through API
   - Same user interface

3.1.2 Step 2: Understanding the API Layer

What is an API Layer?

An API is essentially a **set of endpoints**.

What are Endpoints?

- Special types of functions
- Publicly available on the Internet
- Anyone can access them
- Have unique URLs
- Can be called/hit by anyone

Example Endpoint:

API Endpoint Example

Endpoint Name: `/trains`

Characteristics:

- It's a function (like any other function)
- But it's a *special* function
- Available and present on the Internet
- Has a public URL
- Anyone can access it

URL Example:

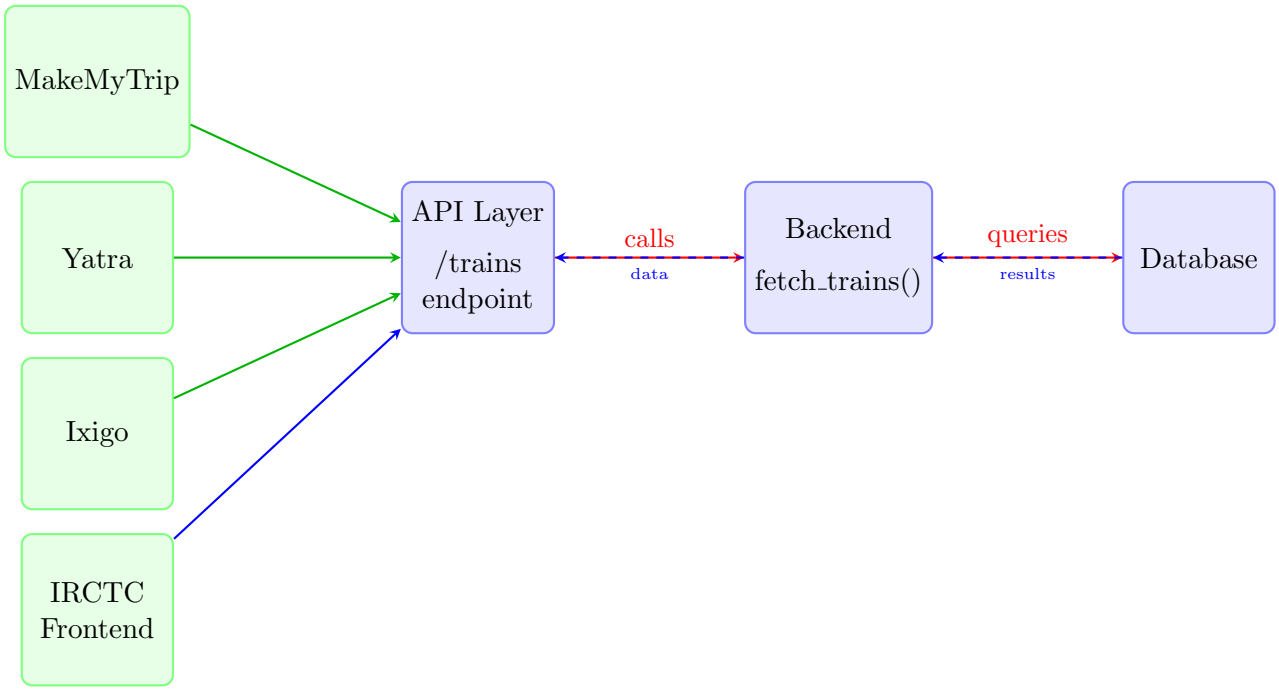
`https://irctc.com/api/trains`

What it does:

- When someone hits this URL
- Behind the scenes, it calls Backend's `fetch_trains()` function
- Backend function queries Database
- Results returned through API

3.2 The Complete API-Based Flow

3.2.1 Visual Representation



3.2.2 Step-by-Step Process

Request Flow:

1. **External Application Makes Request:**
   - MakeMyTrip/Yatra/Ixigo hits API endpoint
   - URL: `https://irctc.com/api/trains`
   - Provides: Station1, Station2, Date
2. **API Receives Request:**
   - API endpoint receives the request
   - Extracts station names and date
   - Prepares to call Backend
3. **API Calls Backend:**
   - API invokes `fetch_trains()` function
   - Passes station names and date
   - Backend function executes
4. **Backend Queries Database:**
   - Backend searches for trains
   - Finds trains between stations on given date
   - Retrieves results from Database
5. **Results Return Path:**

- Database returns data to Backend
- Backend sends data to API
- API formats response (JSON)
- API sends response to requester

6. **Client Receives Response:**
   - MakeMyTrip/Yatra/Ixigo receives data
   - Displays information to their users
   - Mission accomplished!

3.3 Key Changes Made

Two Major Changes

To solve the Monolithic Architecture problem:

Change 1: Decoupling

- Separated Backend from Frontend
- Backend = Independent application
- Frontend = Independent application
- No tight coupling

Change 2: API Layer

- Added API layer in front of Backend
- Made Backend publicly accessible via Internet
- Backend available through API endpoints
- Anyone can access through proper API calls

3.4 Benefits of API Architecture

3.4.1 Problem Solved

External Access:

- Companies can now access IRCTC's data
- No direct database access (secure)
- No direct backend access (controlled)
- Access through API (safe and monitored)

Revenue Generation:

- IRCTC can charge per API request
- MakeMyTrip pays for each query
- Yatra pays for each query

- Ixigo pays for each query
- New business model enabled

3.4.2 Additional Advantages

Security and Control:

- API can implement security checks
- Validate incoming requests
- Prevent malicious data
- Authentication and authorization
- Rate limiting
- Logging and monitoring

API Security Example

```python
# API endpoint can validate requests
def trains_endpoint(station1, station2, date):
    # Security checks
    if not is_valid_station(station1):
        return error("Invalid station")

    if not is_valid_date(date):
        return error("Invalid date")

    # Check API key/authentication
    if not is_authenticated():
        return error("Unauthorized")

    # Only then call backend
    result = backend.fetch_trains(station1, station2, date)
    return format_as_json(result)
```

IRCTC's Own Frontend:

Important note: IRCTC's own Frontend is NOT special!

- IRCTC Frontend is now a separate application
- It also uses the same API
- Same treatment as MakeMyTrip/Yatra/Ixigo
- Goes through API to reach Backend
- No direct Backend access

Important Note

Key Insight:

The API-based architecture treats all clients equally:

- Your own Frontend
- External companies
- Mobile apps
- Third-party services

Everyone goes through the API. This ensures consistency, security, and scalability.

3.5 Making Database Knowledge Available to All

The Big Picture:

- Database information is now accessible to everyone
- Not directly, but through controlled API access
- IRCTC can share data safely
- External parties can use the data
- IRCTC generates revenue
- Win-win for all stakeholders

Business Impact

Before API:

- Data locked in monolithic system
- No way to share with external parties
- Lost revenue opportunities
- Limited ecosystem

After API:

- Data accessible through controlled endpoints
- External companies can integrate
- Revenue generation through API access
- Thriving ecosystem

4 API Protocols and Data Formats

4.1 Revisiting the Definition

Let's look again at the API definition with deeper understanding:

API Definition - Revisited

APIs are mechanisms that enable two software components (such as the frontend and backend of an application) to communicate with each other using a defined set of rules, protocols, and data formats.

Now we understand:

- ✓ How API acts as a connector
- ✓ How it connects two software components
- → Now let's understand: Protocols and Data Formats

4.2 Protocols in API Communication

4.2.1 What is HTTP Protocol?

When communicating over the Internet:

- All web-based communication uses **HTTP protocol**
- HTTP = HyperText Transfer Protocol
- Standard protocol for web applications
- Defines how requests and responses are formatted

HTTP in Our Example

In the IRCTC example:

- MakeMyTrip sends request to API: Uses HTTP
- API communicates with Backend: Uses HTTP
- Backend returns response: Uses HTTP
- Response sent to MakeMyTrip: Uses HTTP

Everything happening over the Internet follows HTTP protocol.

4.3 Data Formats: JSON

4.3.1 The Need for Universal Format

The Challenge:

Consider our API clients:

- MakeMyTrip might be built in **Java**
- Yatra might be built in **Python**

- Ixigo might be built in **PHP**

  The Question:

  When API returns response, what format should it use?

- Java needs to understand it
- Python needs to understand it
- PHP needs to understand it
- Need a universal format!

4.3.2 JSON: The Universal Data Format

JSON - JavaScript Object Notation

What is JSON?

- Universal data format
- Language-independent
- Human-readable
- Machine-parseable
- Industry standard for APIs

Why JSON?

- Java can understand JSON
- Python can understand JSON
- PHP can understand JSON
- Every programming language supports JSON
- Perfect for API responses

4.3.3 JSON Format Example

JSON Response Example

```
# API returns data in JSON format
{
    "trains": [
        {
            "train_number": "12345",
            "train_name": "Rajdhani Express",
            "departure": "10:00 AM",
            "arrival": "08:00 PM"
        },
        {
            "train_number": "12346",
            "train_name": "Shatabdi Express",
            "departure": "02:00 PM",
```

```
            "arrival": "10:00 PM"
        }
    ],
    "status": "success",
    "count": 2
}
```

Important Note

For Python Developers:

If you've worked with Python dictionaries, JSON looks very similar!

```
# Python dictionary
data = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# Almost identical to JSON!
```

We'll see JSON formatting in detail when we write code in FastAPI.

4.4 Key Takeaways

Three Important Points

1. API as Connector:

- Connects two software components
- In our case: IRCTC Backend ¡-¿ External Applications

2. Protocol - HTTP:

- Communication follows HTTP protocol
- Standard for Internet-based communication

3. Data Format - JSON:

- Responses returned in JSON format
- Universal format understood by all languages
- Client can be in any language - no problem!

5 The Second Problem: Multiple Platforms

5.1 The Smartphone Revolution

5.1.1 The Timeline: 2008-2012

Before 2008:

- Nobody had smartphones
- Only basic mobile phones existed
- Desktop/laptop was primary way to access internet

2008-2012:

- Smartphone revolution began
- By 2012, most people had smartphones
- Two major categories emerged:
  - Android phones
  - iPhones (iOS)

5.2 The Multi-Platform Challenge

5.2.1 Business Realization

Companies started realizing:

- Just a website isn't enough
- To expand business opportunities
- Need to be on multiple platforms
- Should build:
  - Website (for desktop users)
  - Android app (for Android users)
  - iPhone app (for iOS users)

The Goal:

- Provide seamless connectivity to users
- Users can access application from any platform
- Same functionality across all platforms
- Better user experience
- Increased user base

5.2.2 Current Reality (2025)

Multi-Platform Reality

Pick any application today:

- Most likely has a website
- Most likely has an Android app
- Most likely has an iPhone app

Examples:

- Social media: Instagram, Facebook, Twitter
- E-commerce: Amazon, Flipkart
- Food delivery: Zomato, Swiggy
- Ride sharing: Uber, Ola
- Entertainment: Netflix, Spotify

All available on Web + Android + iOS

5.3 The Technical Challenge

5.3.1 Different Technology Stacks

The Problem:

Each platform requires different technology:

Platform	Language	Tech Stack
Website	JavaScript/Python/PHP	HTML, CSS, JS
Android App	Java/Kotlin	Android SDK
iPhone App	Swift/Objective-C	iOS SDK

What This Means:

- Completely different tech stack for Android
- Completely different tech stack for iOS
- Completely different tech stack for Web
- Cannot reuse code across platforms

5.3.2 The Monolithic Approach Problem

If Using Monolithic Architecture:

Would need to build:

1. One Monolithic application for Website
2. One Monolithic application for Android app
3. One Monolithic application for iPhone app

Consequences:

Warning

Major Problems with Three Monolithic Apps:

Problem 1: Independence

- All three applications are independent
- Changes must be made to all three
- User comments on website? Must update Android and iOS
- User uploads photo on Android? Must sync to Web and iOS
- Everything needs to be synchronized manually

Problem 2: Team Requirements

- Need three separate development teams
- Web development team
- Android development team
- iOS development team
- Significant additional cost

Problem 3: Maintenance Nightmare

- Three codebases to maintain
- Three sets of bugs to fix
- Three deployments for every feature
- Extremely difficult to keep synchronized

5.4 The API-Based Solution

5.4.1 The Elegant Architecture

Instead of three Monolithic applications, use API architecture:

API-Based Multi-Platform Architecture

Single Backend + Multiple Frontends

Backend (One):

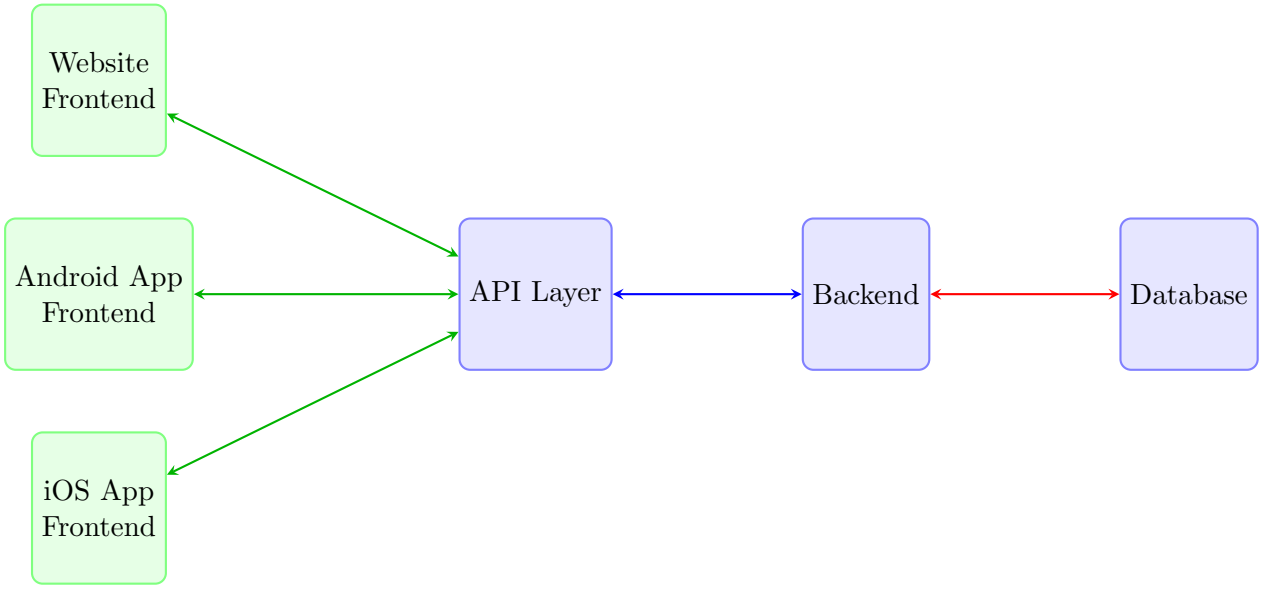
- One Database (single source of truth)
- One Backend application
- One API layer
- Handles all business logic

Frontends (Multiple):

- Website Frontend (separate)

- Android App Frontend (separate)
- iOS App Frontend (separate)
- All connect to same API

5.4.2 Visual Architecture



5.4.3 How It Works

The Flow:

1. **User Makes Request:**
   - Could be from Website, Android, or iOS
   - Request goes to API
2. **API Processes:**
   - API receives request
   - Forwards to Backend
   - Doesn't care which frontend sent it
3. **Backend Processes:**
   - Backend handles business logic
   - Queries Database if needed
   - Processes data
4. **Response Returns:**
   - Database → Backend → API
   - API formats as JSON
   - JSON sent to requesting frontend

5. **Frontend Displays:**
    - Each frontend displays in its own way
    - Website: HTML/CSS rendering
    - Android: Native Android UI
    - iOS: Native iOS UI

5.5 Benefits of API-Based Multi-Platform

5.5.1 Simplified Architecture

- **One Database**: Single source of truth for all data
- **One Backend**: All business logic in one place
- **One API**: Single gateway for all platforms
- **Multiple Frontends**: Platform-specific UIs only

5.5.2 Development Benefits

No Need for Separate Backends:

- Don't need three different backend implementations
- Same backend serves all platforms
- Write business logic once, use everywhere

No Need for Multiple Databases:

- Don't maintain three databases
- All data in one place
- Automatic synchronization
- Data consistency guaranteed

Frontend Focus:

- Frontend teams only worry about UI/UX
- Each platform optimized independently
- Same data, different presentation

5.5.3 Real-World Examples

Industry Implementation

Companies Using This Architecture:

Google:

- One backend API
- Multiple frontends: Web, Android, iOS, Desktop
- Same Gmail data everywhere

Uber:

- One backend system
- Passenger apps: Web, Android, iOS
- Driver apps: Android, iOS
- All hitting same APIs

Zomato:

- Single database of restaurants
- One backend with APIs
- Website + Android + iOS apps
- Consistent experience everywhere

Important Note

Today, if you examine any major company (Google, Uber, Zomato, etc.), you'll find:

- Single database
- Backend connected to database
- Backend accessible via API
- Multiple frontends consuming the API

This is the standard architecture for modern applications.

5.6 Summary: Two Problems Solved

Problem 1: External Access

Challenge:

- External companies needed access to data
- Monolithic architecture prevented sharing
- No way to monetize data safely

Solution:

- Decouple Backend from Frontend
- Add API layer
- External parties access through API
- Secure, controlled access
- Revenue generation possible

Problem 2: Multiple Platforms

Challenge:

- Need Website, Android, and iOS apps
- Different tech stacks for each
- Maintaining three monolithic apps too complex
- Synchronization nightmare

Solution:

- Single Backend + API
- Multiple independent Frontends
- All Frontends use same API
- Simplified architecture
- Easy maintenance
- Automatic synchronization

6 APIs in Machine Learning Context

6.1 Software vs ML: The Key Difference

Now let's understand how these API principles apply specifically to Machine Learning.

6.1.1 The Core Difference

Software Applications	**ML Applications**
Most important: **Database**	Most important: **ML Model**
Data stored in database	Data learned by model
Query database for information	Query model for predictions
Backend interacts with database	Backend interacts with model

Key Insight

In Machine Learning:

Replace "Database" with "ML Model" and the entire architecture remains the same!

- Database → ML Model
- Query → Prediction Request
- Results → Predictions
- Everything else identical

6.2 ML Model Deployment Journey

6.2.1 The ML Development Process

1. **Model Training:**
   - Train model on large dataset
   - Could be ML, DL, or Generative AI
   - Iterate until good results
2. **Model Evaluation:**
   - Model starts giving good results
   - Metrics look promising
   - Ready for production
3. **Deployment Decision:**
   - Want to present model to world
   - Need to monetize
   - Build application around model
4. **Application Building:**
   - Build around the model
   - Create user interface
   - Enable model access

6.3 Case Study: ChatGPT

Let's understand ML APIs through the world's most famous AI product.

6.3.1 ChatGPT Background

What is ChatGPT?

- Built by OpenAI
- Powered by GPT models (LLMs)
- Most famous AI product currently
- Uses Large Language Models

OpenAI's Process:

1. Trained GPT model on massive data
2. Model started producing good results
3. Decided to monetize the model
4. Best way: Present it via a website

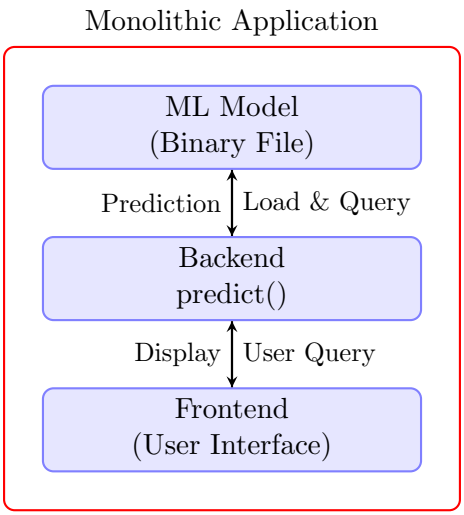
The Goal:

- Users visit website
- Users ask questions
- Questions sent to GPT model
- Model generates responses
- Responses displayed to users

6.4 Pre-API ML Application Architecture

6.4.1 The Monolithic ML Approach

Before APIs, ML applications were built as monolithic systems:



6.4.2 Components Breakdown

1. ML Model (Binary File):

- Trained model saved as file
- Could be .pkl, .h5, .pth, etc.
- Not a database - it's a file
- Can be loaded in any programming language

2. Backend:

- Functions to interact with model
- `predict()` function example
- Takes user data
- Loads model
- Sends data to model
- Returns prediction

Backend Prediction Function

```python
def predict(user_query):
    """
    Load ML model and get prediction
    """
    # Load the trained model
    model = load_model('chatbot_model.pkl')

    # Get prediction from model
    response = model.generate_response(user_query)

    # Return the response
    return response
```

3. Frontend:

- User interface
- Similar to ChatGPT interface
- Text input box for questions
- Submit button
- Display area for responses

6.4.3 The Monolithic ML Flow

1. **User Interaction:**
   - User opens website (Frontend)
   - Types question in text box
   - Clicks Submit
2. **Request Processing:**
   - User query sent to Backend
   - `predict()` function called
   - Function receives query
3. **Model Inference:**
   - Backend loads ML model
   - Sends query to model
   - "What should the answer be?"
4. **Response Generation:**
   - Model generates answer
   - Based on its training
   - Returns answer to Backend
5. **Display Result:**
   - Backend sends answer to Frontend
   - Frontend displays to user
   - User sees the response

6.4.4 The Problem with Monolithic ML Apps

Warning

Same Issues as Regular Applications:

Tight Coupling:

- Everything in one folder/application
- Frontend, Backend, Model all together
- One unified monolithic application
- Difficult to scale
- Difficult to maintain

Cannot Share Model:

- External applications cannot access
- Cannot monetize model separately
- Limited to own frontend only

- No ecosystem development

6.5 Real-World ML Business Scenario

6.5.1 The ChatGPT Opportunity

When ChatGPT launched, many companies realized its potential:

Business Use Cases

1. Swiggy/Zomato - Food Delivery:

- Use automated chatbots for customer service
- But chatbots weren't very good
- Poor responses, bad user experience
- Saw ChatGPT's capabilities
- Thought: "If we could use ChatGPT for our chatbot, user experience would improve dramatically"

2. Amazon - E-commerce:

- Thousands of product reviews
- Users don't read all reviews
- Thought: "What if we use GPT to summarize reviews?"
- Show users a condensed summary
- Improved shopping experience

3. Document Analysis Companies:

- Need question-answering on documents
- Thought: "What if we build RAG system with GPT?"
- Use GPT model for document Q&A
- Better than traditional search

6.5.2 The Business Problem

The Desire:

Many companies want to use ChatGPT's capabilities:

- Chatbot improvements
- Review summarization
- Document Q&A (RAG systems)
- Various other applications

The Blocker:

Warning

Cannot Access the Model:

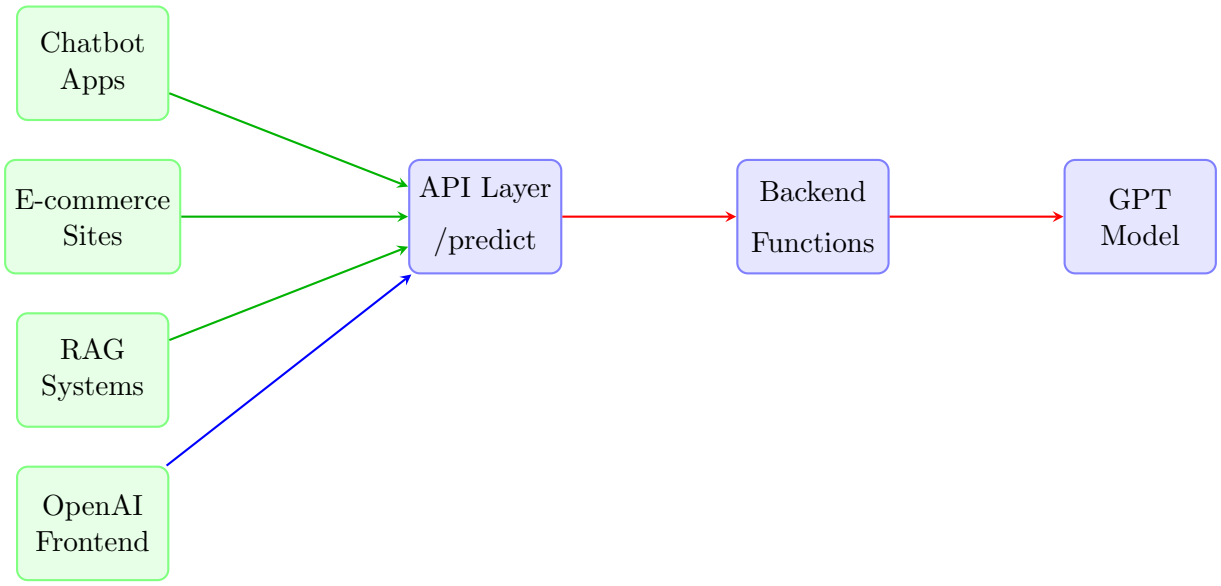
Because OpenAI's application is Monolithic:

- Cannot directly access the GPT model
- Cannot directly access the Backend
- Model and Backend are tightly coupled
- No way for external apps to integrate
- Same problem we saw with IRCTC!

6.6 The API Solution for ML

6.6.1 Decoupling the ML Application

The solution is identical to regular software - use APIs!



6.6.2 The New Architecture

Components:

1. **ML Model (Independent):**
   - GPT model as separate entity
   - Saved as binary file
   - Can be loaded and queried
2. **Backend (Independent):**
   - Separate application

- Contains functions to interact with model
- Loads model and gets predictions
- Not connected to Frontend directly

3. **API Layer (Gateway):**
   - Sits in front of Backend
   - Makes Backend publicly accessible
   - Endpoints available on Internet
   - Anyone can hit these endpoints
4. **Frontend (Independent):**
   - OpenAI's ChatGPT interface
   - Separate application
   - Communicates through API only
5. **External Applications:**
   - Other companies' applications
   - Chatbots, e-commerce sites, RAG systems
   - All communicate through API

6.6.3 Complete ML API Flow

1. **External App Makes Request:**
   - Zomato's chatbot needs a response
   - Hits API endpoint with user query
   - `POST /api/predict`
2. **API Receives Request:**
   - API endpoint receives query
   - Validates request
   - Forwards to Backend
3. **Backend Processes:**
   - Backend function called
   - Loads GPT model
   - Sends query to model
4. **Model Generates Response:**
   - GPT model processes query
   - Generates answer
   - Returns to Backend
5. **Response Returns:**
   - Backend sends response to API

- API formats as JSON
- JSON sent back to requester

6. **Client Uses Response:**

- Zomato's chatbot receives response
- Displays to their users
- Improved user experience!

6.7 ML API Architecture Comparison

Software APIs	**ML APIs**
Database stores data	ML Model stores learned patterns
Backend queries database	Backend queries model
Returns data results	Returns predictions
API exposes data access	API exposes prediction access
HTTP protocol	HTTP protocol
JSON data format	JSON data format
Multiple frontends access	Multiple apps access

Key Observation

The Architecture is Identical!

From an architecture perspective, ML APIs and Software APIs are exactly the same. The only difference:

- Software: Backend interacts with Database
- ML: Backend interacts with Model

Everything else (protocols, data formats, API structure) remains unchanged.

6.8 Benefits of ML APIs

6.8.1 For Model Creators (OpenAI)

Monetization:

- Charge per API request
- Pricing tiers based on usage
- Sustainable business model
- Revenue from multiple clients

Scalability:

- Single model serves thousands of clients
- Centralized model updates
- Improved model = All clients benefit
- Easy to deploy improvements

Security:

- Model not directly accessible
- Intellectual property protected
- Controlled access through API
- Rate limiting and monitoring

6.8.2 For Clients (Companies Using API)

Easy Integration:

- Don't need ML expertise
- Don't need to train models
- Just make API calls
- Focus on their core business

Cost Effective:

- Pay per use
- No infrastructure needed
- No model training costs
- No maintenance overhead

Always Updated:

- Model improvements automatic
- No need to retrain
- Latest capabilities always available
- Continuous improvement

6.9 Multi-Platform ML Deployment

Just like software, ML applications need multi-platform support.

6.9.1 The Amazon Recommender Example

Scenario:

Amazon builds a recommender system:

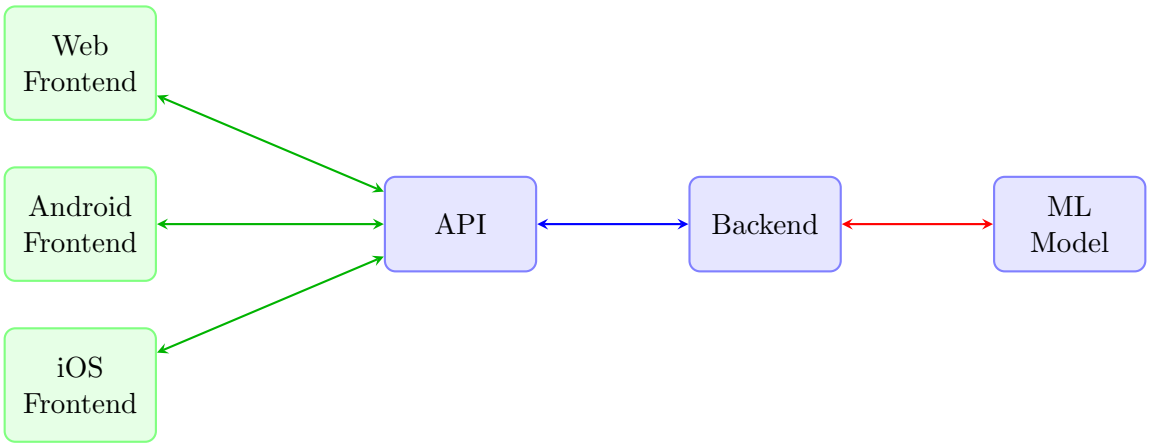
- ML model trained on user behavior
- Recommends products based on viewing history
- Needs to work on all platforms

Without APIs (Monolithic):

Would need:

- Separate recommender for Website
- Separate recommender for Android app
- Separate recommender for iOS app
- Three times the work!

With APIs (Decoupled):



Benefits:

- One ML model
- One Backend
- One API
- Three separate Frontends
- All frontends use same recommender
- Consistent recommendations everywhere

6.10 Industry Standard ML Architecture

Current Industry Practice

Standard ML Deployment Architecture:

When any company deploys an ML-based application:

1. Train ML/DL/Generative AI model
2. Build Backend to interact with model
3. Create API layer over Backend
4. Develop platform-specific Frontends
5. All Frontends consume same API

This is followed by:

- Startups building AI products

- Large tech companies (Google, Microsoft)
- ML-focused companies (OpenAI, Anthropic)
- Any company deploying ML models

6.11 Summary: APIs in ML Context

Key Takeaways

1. Similar Architecture:

- ML APIs follow same pattern as Software APIs
- Only difference: Model instead of Database
- Same protocols (HTTP)
- Same data formats (JSON)

2. Decoupling Benefits:

- Model accessible to external applications
- Multiple platforms can use same model
- Easy integration and deployment
- Monetization opportunities

3. Industry Standard:

- Every ML product uses API architecture
- From ChatGPT to recommendation systems
- Essential for production deployment
- Critical skill for ML engineers

7 Summary and Next Steps

7.1 What We Learned Today

7.1.1 Main Topics Covered

1. What are APIs?

- Mechanisms enabling software communication
- Act as connectors between applications
- Follow defined protocols and data formats
- Essential for modern application architecture

2. Why APIs Exist?

- Problem 1: External data access
  - Monolithic apps couldn't share data
  - APIs enable controlled external access
  - Monetization opportunities
- Problem 2: Multi-platform support
  - Need for Web, Android, iOS apps
  - Single Backend + API + Multiple Frontends
  - Efficient architecture

3. APIs in Machine Learning

- Same architecture principles apply
- Model replaces Database
- Backend interacts with Model
- API exposes prediction functionality
- Standard for ML deployment

7.2 Key Concepts Recap

7.2.1 Monolithic vs API Architecture

Monolithic	**API-Based**
Everything in one application	Separate components
Tightly coupled	Loosely coupled
Cannot share externally	External access possible
Difficult to scale	Easy to scale
One platform only	Multi-platform support
Limited flexibility	Highly flexible

7.2.2 API Communication

Protocol: HTTP

- Standard Internet protocol
- Request-response model
- Used by all web-based APIs

Data Format: JSON

- Universal data format
- Language-independent
- Human-readable
- Machine-parseable

7.3 Real-World Applications

Use Cases Discussed

Software Domain:

- IRCTC train information sharing
- Multi-platform applications (Web, Android, iOS)
- Third-party integrations

ML Domain:

- ChatGPT/GPT model access
- Recommender systems (Amazon)
- Chatbot improvements (Zomato)
- Review summarization (E-commerce)
- RAG systems (Document Q&A)