Reactive Programming

Reactive Programming becomes very easy to understand when you map it to something from real life.

Let's use a **food delivery app** — something everyone already understands.

Scenario: Tracking Your Food Order

When you place an order, many things happen:

- Restaurant accepts order
- Food is prepared
- Delivery partner picks it up
- Driver location updates
- Order delivered

These are **events happening over time**, not a single response.

Traditional (Imperative) Approach

In a non-reactive system, your app might repeatedly ask the server:

"Is my order ready?"
"Has the driver moved?"
"Any updates?"

This is called **polling**.

Problems:

Wasteful network calls
Delays between checks
Poor scalability

Reactive Approach

In Reactive Programming, you **subscribe to updates**.

Your app tells the system:

“Notify me whenever something changes.”

Now updates are pushed automatically.

How It Works Conceptually

Reactive Programming:

Tracking Your Food Order

Instead of Repeatedly Asking:

Is it ready yet? *Where's my driver?* *Any updates?*

You Get Real-Time Updates:

Order Events Stream

Restaurant Accepted → Driver Assigned → Driver Location Update → Order Delivered

Real-Time Notifications

Live Map Updates

You Subscribe to Updates, React as They Happen!

Instead of asking repeatedly:

1. Server publishes order events
2. App subscribes to the order stream
3. Each update arrives instantly
4. UI reacts automatically

What Actually Happens Behind the Scenes

Think of your order as a **stream of events**:

Order Placed
→ **Restaurant Accepted**
→ **Cooking Started**
→ **Driver Assigned**
→ **Driver Location Changed**
→ **Delivered**

Each step is emitted as a new value in the stream.

Your app reacts:

- **New status** → Update progress bar
- **Driver moved** → Update map
- **Delay event** → Show message

Why This is Reactive Programming

Reactive Programming is about:

Handling **multiple values over time**
Responding to **events as they occur**
Avoiding blocking & constant checking

No loops, no repeated requests.

Another Real-Time Example: Live Ride-Sharing App

When you open a ride-sharing app:

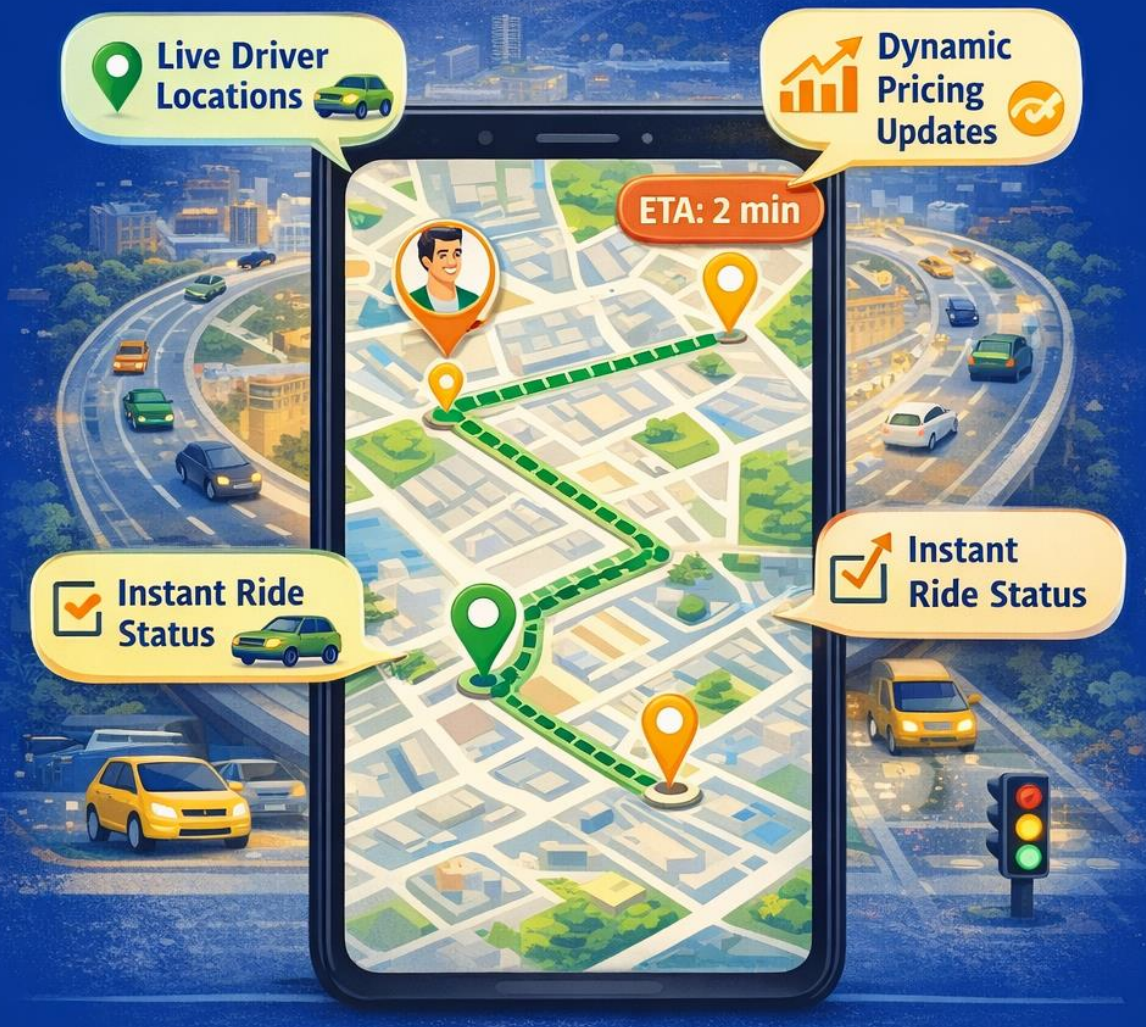
- Drivers move continuously
- Prices change dynamically
- Ride requests appear
- Traffic conditions update

Your screen updates **without refreshing**.

That’s reactive behavior.

Reactive Programming:

Live Ride-Sharing App



No Refreshing. Drivers Move. Prices React. Traffic Changes.

All in Real-Time!

Core Insight

Reactive Programming fits naturally when:

Data changes frequently
Updates must be real-time
Many users consume the same events
Blocking would be harmful

Perfect for:

- Chats
- Notifications
- IoT sensors
- Financial data
- Live dashboards

One-Sentence Mental Model

Reactive Programming =

“Subscribe to what you care about, then react whenever it changes.”

Real-Time Example: Live Traffic Monitoring System

Imagine you are building a **Google-Maps-like traffic screen**.

Cars are constantly moving. Traffic conditions change every second.

Imperative Programming Approach

In an imperative system, the app must **continuously ask for updates**.

Logic:

Loop forever:

Request latest traffic data from server
Wait for response
Update map
Sleep for 3 seconds

What happens:

- App keeps sending requests
- Server keeps responding
- Updates only occur at intervals
- Resources wasted even when nothing changes

Problems with Imperative Style

Unnecessary network calls
Delayed updates (traffic changed immediately, UI waits)
Heavy server load
Poor scalability under many users

Visualizing Imperative Flow

System behavior:

"*Anything new? Anything new? Anything new?*"

Reactive Programming Approach

In reactive programming, the model flips.

Instead of asking repeatedly…

The app subscribes to traffic updates.

Logic:

Subscribe to traffic event stream
Whenever traffic changes → Update map instantly

What happens:

- **Server pushes updates only when changes occur**
- **No constant requests**
- **UI updates immediately**
- **System stays efficient**

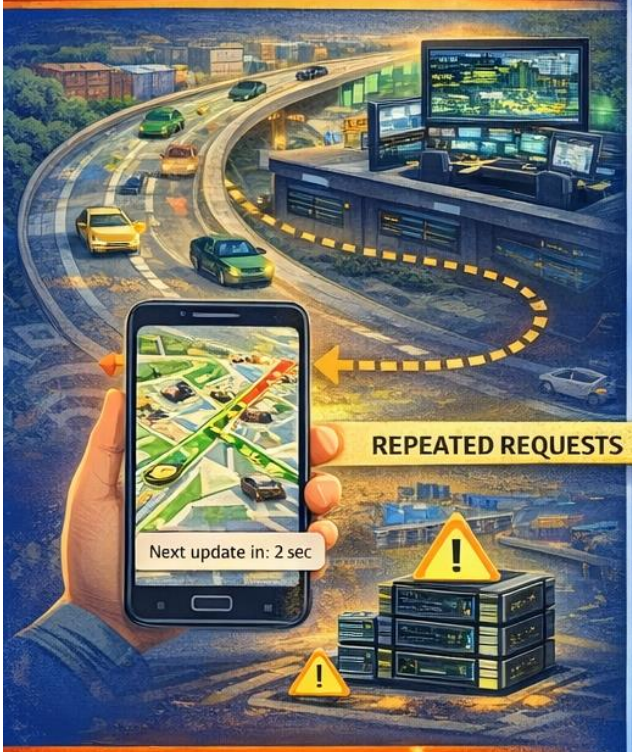
Benefits of Reactive Style

Real-time updates
No wasted calls
Efficient resource usage
Handles thousands/millions of users better

Visualizing Reactive Flow

Live Traffic
Monitoring System

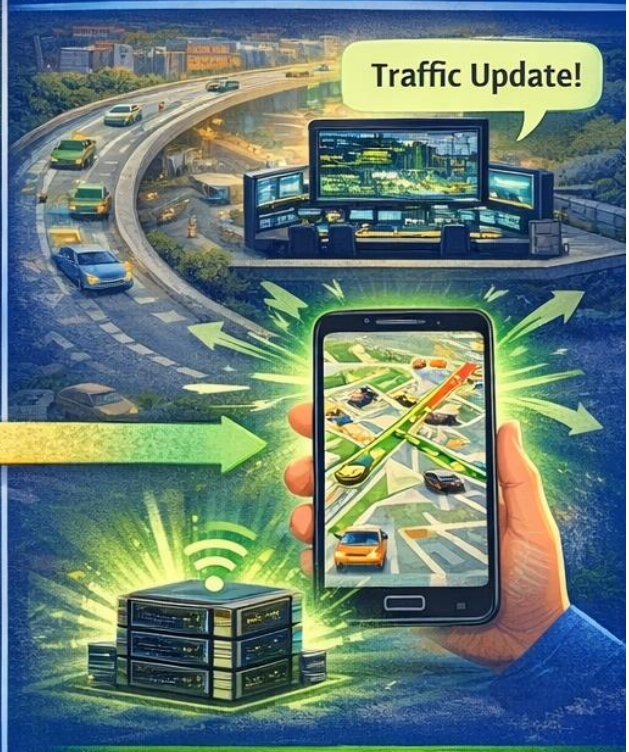
IMPERATIVE PROGRAMMING



Asks Repeatedly
Request Latest Traffic Data

- Wasteful Requests
- Delayed Updates
- Heavy Load

REACTIVE PROGRAMMING



Gets Updates When Traffic Changes

- Real-Time Updates
- Efficient Updates
- Light Load

IMPERATIVE vs REACTIVE

Asks Repeatedly
Request Latest Traffic Data

Gets Updates When Traffic Changes
Listen to Traffic Event Stream

System behavior:

"Traffic changed → Notify subscribers immediately."

Traffic systems are ideal for reactive programming because:

Data changes continuously
Users need instant updates
Polling would overload servers
Events drive UI updates

Exactly how:

- Live maps
- Stock markets
- Chats
- IoT sensors
- Ride-sharing apps

work internally.

Core Insight

Imperative Programming

Pull model → Ask for data repeatedly

Reactive Programming

Push model → React to data when it changes

Synchronous vs. Asynchronous Execution

The easiest way to understand this is through a **very familiar real-world situation**.

Real-Time Example: Ordering Coffee at a Café

Synchronous Execution (Blocking Behavior)

Imagine a café where:

- There is **one barista**
- They serve **one customer at a time**
- Next customer must **wait until current order finishes**

Flow:

Customer 1 orders → Barista prepares coffee → Customer 2 waits

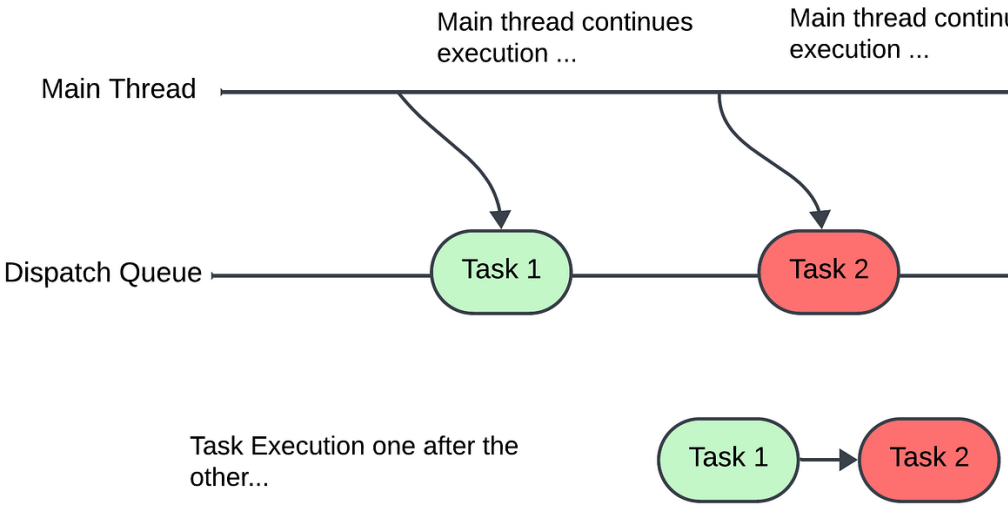
Nothing else happens until the task completes.

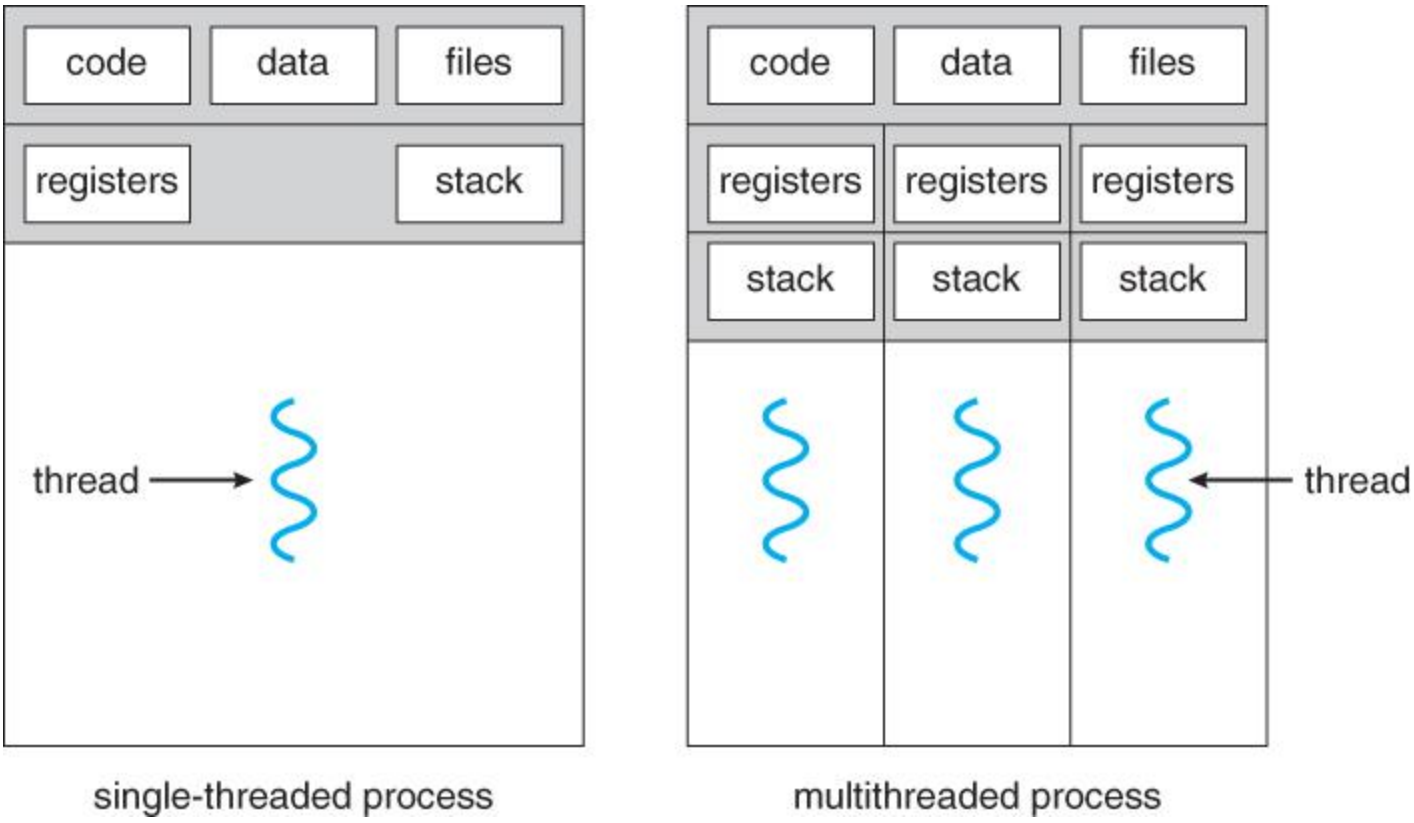
Characteristics of Synchronous Execution

Tasks run sequentially
Caller must wait
Idle time wastes resources
Slow under heavy load

Visualizing Synchronous Model

Serial Async





System behavior:

"Wait until this finishes, then continue."

Asynchronous Execution (Non-Blocking Behavior)

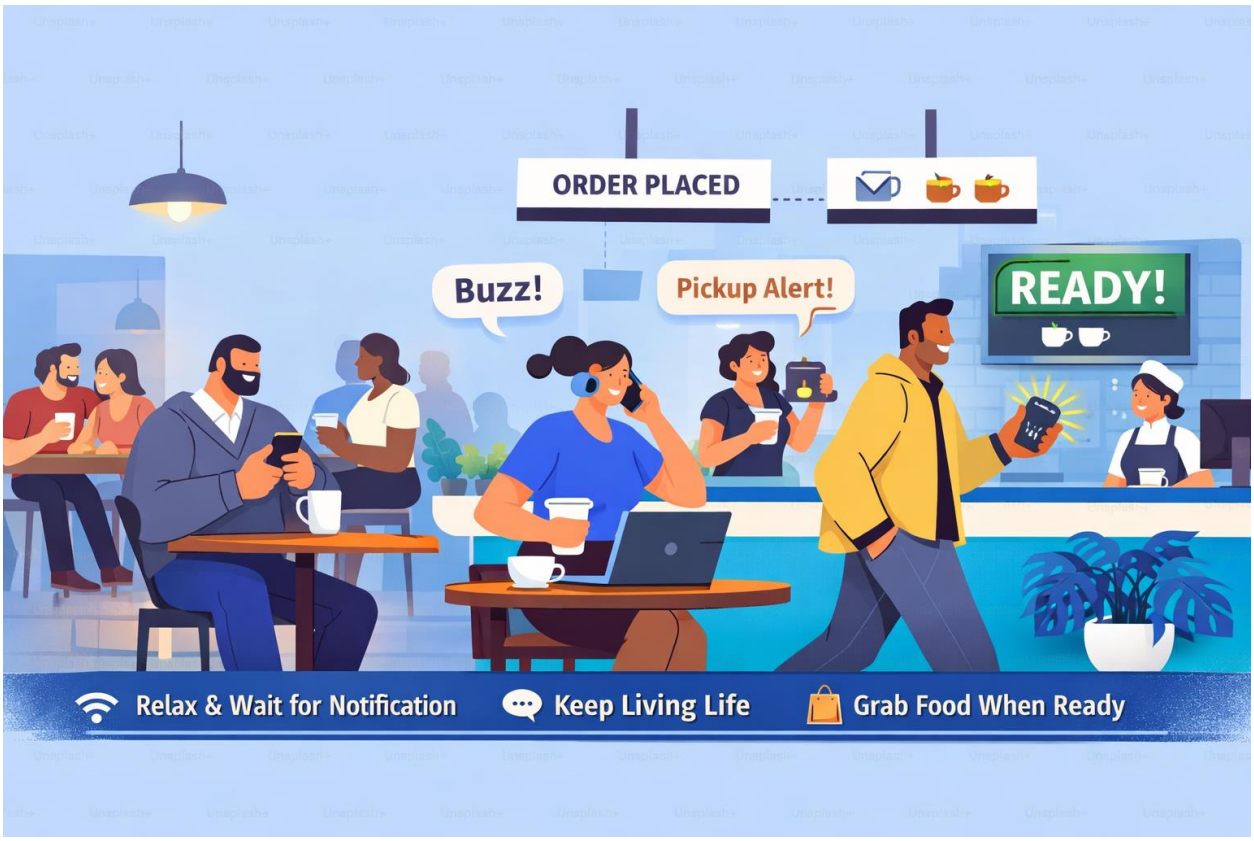
Now imagine a smarter café:

- **Orders are taken instantly**
- **Multiple coffees prepared in parallel**
- **Customers receive notification when ready**

Flow:

Customer 1 orders → Preparation starts
Customer 2 orders → Preparation also starts
No waiting required

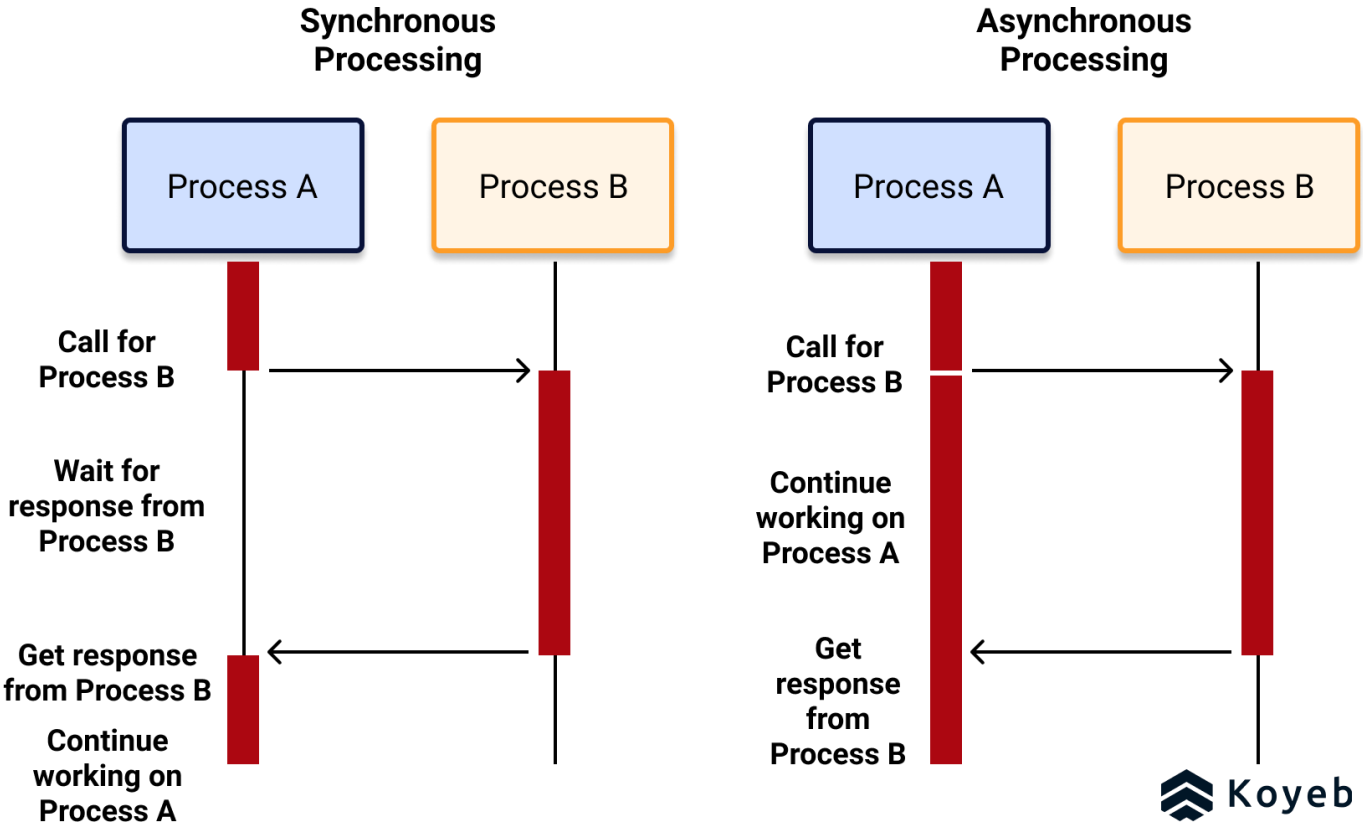
Work continues without blocking.



Characteristics of Asynchronous Execution

Tasks overlap
No forced waiting
Better performance
Better scalability

Visualizing Asynchronous Model



System behavior:

"Start task → Continue other work → Handle result later."

Why This Matters in Software

Consider a mobile app calling an API.

Synchronous Call

Request data → UI freezes → Wait → Response → Continue

Bad user experience.

Asynchronous Call

Request data → UI remains responsive → Response handled later

Smooth & scalable.

Core Difference

Synchronous Execution

Caller is blocked.

Asynchronous Execution

Caller is free to continue.

One-Line Mental Model

Synchronous = *"Do this first, nothing else allowed."*
Asynchronous = *"Start this, continue doing other things."*