WebSocket vs. HTTP communication protocols

Cameron Pavey

Tutorial Type: Basics   Reading Time: 15 min   Building Time: N/A

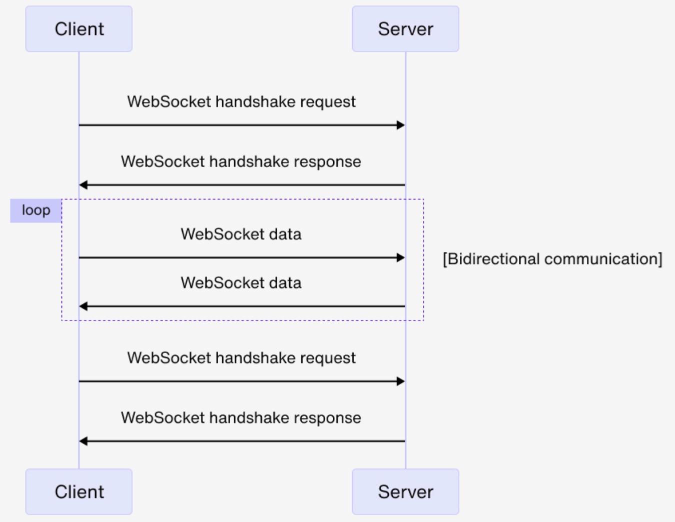
WebSocket vs. HTTP in a nutshell

Many of your daily activities on the internet—from ordering food to looking up a fact to speaking to a doctor online—are enabled by WebSocket or HTTP communication protocols. As a developer, when building an app, which of these communications protocols should you use? What are the differences when we compare WebSocket vs. HTTP? In a nutshell, WebSocket, a full-duplex communication protocol, is relatively newer and is well suited to real-time applications such as in-app chat, notifications, and voice or video calls. On the other hand, HTTP, a half-duplex communication protocol, has been around for some time and has been the basis of websites since its debut.

In this comparison, you'll learn more about these two communication protocols, their similarities and differences, as well as understand when to choose one over the other. Let's dive in!

About the WebSocket connection

The WebSocket protocol describes how a client and server communicate in full-duplex channels. In other words, both the client and server can send and receive data simultaneously over a long-lived connection. This type of communication has less overhead than HTTP polling, giving an application several advantages in real-time functionality.



Client
Server
WebSocket handshake request
WebSocket handshake response
loop
WebSocket data
WebSocket data
[Bidirectional communication]
WebSocket handshake request
WebSocket handshake response
Client
Server

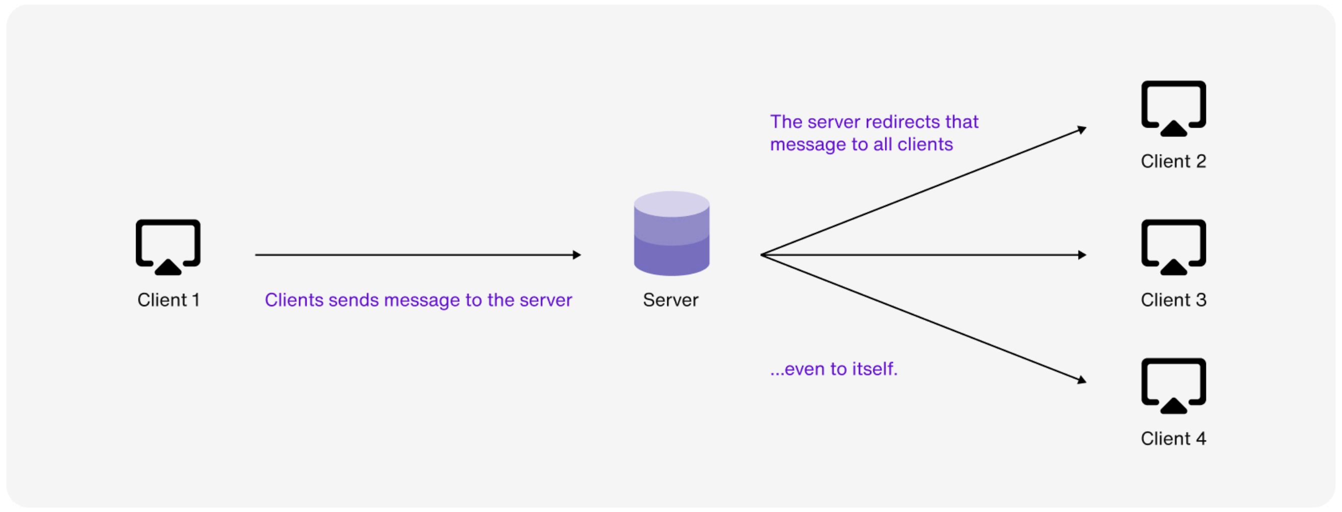
Advantages of a WebSocket connection

Bidirectional communication

Because both sides of the connection can send messages whenever they want, a WebSocket connection is an excellent choice when you need to move a lot of data back and forth quickly.

Imagine a simple chat room connecting multiple clients. If a WebSocket server moderates their conversation, a client sends a message to the server, which then immediately relays it to all other connected clients. As far as the users are concerned, they can send messages to each other in real time.

This is a simple version of what this looks like:


The server redirects that message to all clients
Client 2
Client 1
Clients sends message to the server
Server
Client 3
...even to itself.
Client 4

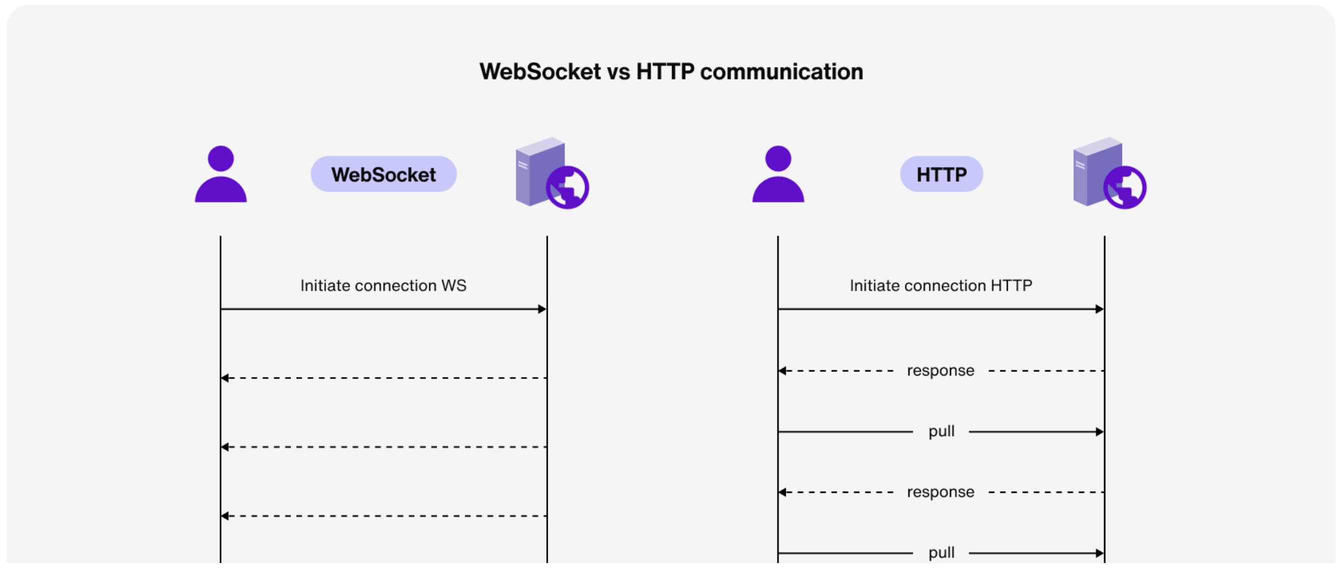
Adapted from [Stack Overflow]()

Lower latency

A common pattern in an HTTP connection for relatively high-frequency data fetching is *polling*, where the client periodically requests new server data. Perhaps the biggest drawback of this communication method is latency—you have to compromise between frequent or long-running requests and high latency.

With a WebSocket connection, data is sent as soon as it's available. The client doesn't need to keep requesting it. The result is much lower latency with a fraction of the overhead and network traffic.

WebSocket vs. HTTP communication diagram



Adapted from source

Persistent connections

With the traditional HTTP connection, the client makes a request, and after the server sends its response, the connection is closed. If the client needs more data, they have to open a new connection.

Note that although HTTP/1.1 introduced persistent connections that allow for the reuse of the TCP/IP connection, this mental model is still helpful and mostly accurate.

With a WebSocket connection, the client can open and use a single connection for all their WebSocket communications with the server. This persistent connection allows for low latency, bidirectional messages.

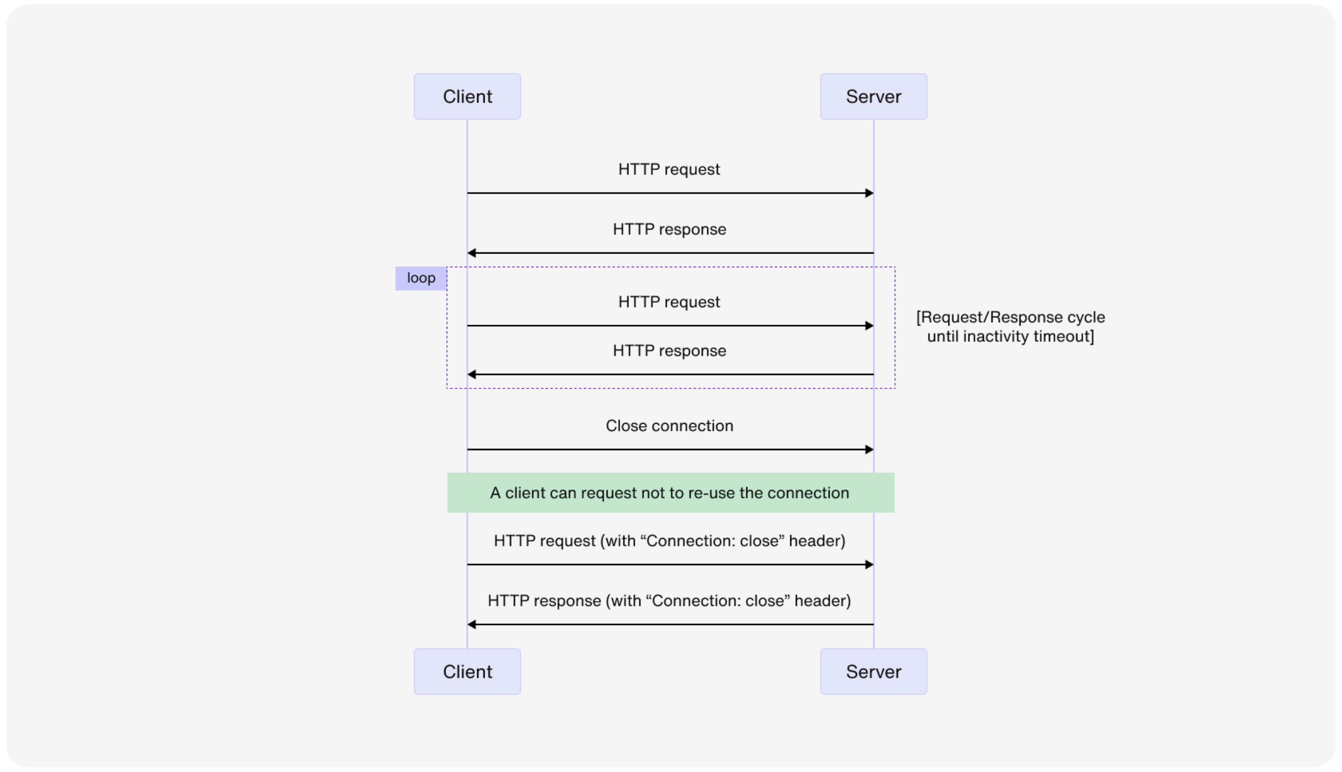
This WebSocket connection can also be *stateful*. An HTTP connection is stateless—this means that each request is handled in isolation, with no retention of information about the requests that came before it. WebSocket, on the other hand, is stateful thanks to its persistent connection.

Whether or not an application takes advantage of stateful capability is entirely up to the developer and how they use their WebSocket connection.

About the HTTP connection

The HTTP protocol was designed as a request-response protocol. A client, such as a browser, would send a *request* to a web server, and the web server would reply with a *response* containing the resources corresponding to the request, such as HTML and CSS files. While HTTP connections are open, they are only half-duplex, meaning communication only goes one way. Once a response has been received, the connection is often closed.

HTTP/1.1 introduced persistent connections that reuse the TCP/IP connection, which allows for some performance improvements. However, the specifics of these persistent connections vary from server to server, and in most cases, they're closed eventually based on an inactivity timeout. So while it's a welcome addition to HTTP connection functionality, this is still not a direct comparison to a WebSocket connection.


Client
Server
HTTP request
HTTP response
loop
HTTP request
HTTP response
[Request/Response cycle until inactivity timeout]
Close connection
A client can request not to re-use the connection
HTTP request (with "Connection: close" header)
HTTP response (with "Connection: close" header)
Client
Server

The HTTP protocol has been very good at what it was built for: responding to requests. However, it wasn't built to handle the real-time communication use cases, such as chat or live event streaming, that people expect today.

Even so, HTTP still has several advantages over the WebSocket protocol.

Advantages of an HTTP connection

Simplicity and ubiquity

The staying power of HTTP connections comes from its widespread adoption and its straightforward accessibility. Between the three major versions of HTTP, virtually all web servers and web browsers can leverage the protocol in some form:

- **HTTP/1.1.** Around 35 percent of sites are still using HTTP/1.1 or below.
- **HTTP/2.** Used by 39.3 percent of all sites.
- **HTTP/3.** Used by 25.7 percent of all sites.

Stateless nature and caching support

Because HTTP requests are stateless and self-contained, a website's performance might benefit from caching responses, especially when dealing with static content and assets. Caching can take place at various levels:

- **In the browser:** This eliminates the need to contact the server at all.
- **At the edge:** This uses a server closer to the user's geographic location, as with CDNs.
- **On the server:** This allows the server to avoid expensive recalculations if the result is the same each time.

WebSocket messages cannot be cached as easily as HTTP responses, given that they're stateful and usually context-sensitive. These messages would change too often for caching to be helpful in most cases.

Robust security mechanisms

The HTTP protocol's ubiquity means it has been the subject of multiple initiatives to improve its security posture.

HTTPS

The original HTTP protocol is lacking in one important respect—request and response messages are not encrypted and are relatively easy for malicious actors to intercept and read.

This problem is mitigated by HTTPS, a variant of HTTP that uses Transport Layer Security (TLS) or Secure Sockets Layer (SSL) to encrypt requests and responses. A malicious actor might be able to intercept your packets, but they won't be able to read their content, thanks to this encryption.

HTTP Strict Transport Security

Another HTTP-related security mechanism is HTTP Strict Transport Security (HSTS). HSTS allows servers to specify policies to help prevent common security problems, such as MITM attacks, protocol downgrade attacks, and cookie hijacking.

A site can leverage HSTS by returning the appropriate header over an HTTPS connection, like so:

Copy code

```
Strict-Transport-Security: max-age=31536000;
```

gistfile1.txt hosted with ❤ by GitHub view raw

When configured correctly, HSTS ensures that the browser will always request the HTTPS variant of the site, even if a user has clicked a standard HTTP link. As a result, the user has a layer of security that protects them from many easy-to-mitigate attacks.

WebSocket vs. HTTP: Choosing the suitable protocol

Before you rally around WebSocket or HTTP protocols, consider what you're building and why. Note that each communication protocol excels in several areas where the other typically falls short.

Technical tradeoffs between a WebSocket vs. HTTP connection

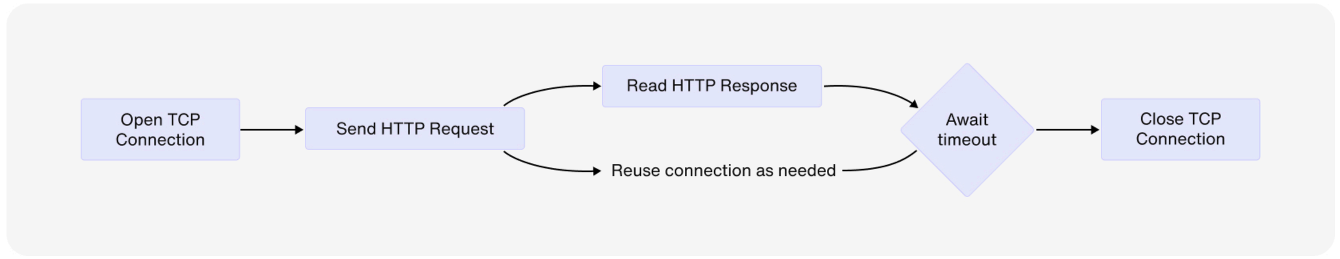
Understanding the technical tradeoffs between these two communication protocols can give you insight as to which one is the best fit for your project.

Connection setup and management

Consider how you need connections to be established and managed over time.

In the case of WebSocket, the persistent connection is established by a handshake between the client and server. It's kept open for the duration of the session, even if there is a significant delay between messages.

With HTTP, connections are established with a handshake and then used for the request-response cycle. HTTP/1.1 allows for the same TCP/IP connection to be reused for multiple request-response pairs, which reduces overhead and improves latency, but not to the same extent as WebSocket. The connection will still close in a relatively short period, ranging anywhere from a few seconds to several minutes.



Data transmission and encoding

Consider how you want data to be transmitted.

A WebSocket connection uses full-duplex two-way communication—either side of the connection can send messages whenever they want. An HTTP connection uses half-duplex communication; only one party can communicate at a time, and the server's message is always in response to a request from a client.

Both WebSocket and HTTP can send data encoded in text-based formats like JSON, XML, and plain text, as well as binary-encoded data.

Error handling and recovery

Consider which error handling methods would be least impactful to users.

A WebSocket connection can fail for various reasons, including errors in your application code. Clients are sent an error event that they can listen for, and you can handle the error in this listener however you see fit. However, if your WebSocket server is running inside your application code, fatal errors at the application level can dramatically impact the ability of your app to implement graceful error handling.

An HTTP connection can, of course, experience similar circumstances, but certain common architectures can provide benefits when it comes to error handling. HTTP specifies a range of status codes that servers can respond with to broadly indicate whether the request has been successful or not. The 4xx and 5xx ranges are reserved for client and server errors, respectively.

Consider a web application where requests are handled through NGINX as the web server and PHP as the dynamic backend language. Let's say something in the application logic results in a fatal error or process termination. This doesn't affect NGINX's ability to serve a response to the client, which would most likely be an HTTP 503 - Service Unavailable message.

Of course, this separation does depend on your application's architecture. Consider a similar situation where your application and web server are implemented together in the same process, such as a Node.js Express app. A fatal error here will also terminate the web server, limiting the usefulness of the error that the client will receive.

Scalability

Consider your application's resource consumption needs.

WebSocket connections are designed to be highly efficient at what they do. They're event-driven—messages are only sent when there is something to send a message about.

An HTTP connection can achieve something akin to real-time functionality through long polling, where requests are sent and held open until there is something to respond with. This rough approximation of real-time communication has some limitations, especially at scale. HTTP requests can't be held open indefinitely, which means that the client will need to periodically open a new long polling request. Over time, the overhead of processing all of these long-lived HTTP requests adds up.

With HTTP streaming, a connection is held open indefinitely to facilitate a continuous data stream. This is conceptually similar to WebSocket, but it's still performed over HTTP and is still one-way—the client cannot send messages to the server via HTTP streaming.

Performance considerations of a WebSocket vs. HTTP connection

Consider your application's performance expectations.

Thanks to persistent connections, WebSocket benefits from reduced overhead and latency. This leads to better performance, faster real-time updates, and less processing power spent on things like the HTTP three-way handshake, and HTTP-specific application code for managing incoming requests and authentication/authorization.

Because HTTP typically has to deal with multiple connections over a session's lifespan, it will naturally spend more time and resources compared to WebSocket.

Security of a WebSocket vs. HTTP connection

Consider which is easiest for you to secure.

WebSocket and HTTP connections are similar regarding security considerations. Both have insecure and secure variants, and both can fall victim to several common attacks if not adequately secured. There are also attacks specific to each protocol that you need to be aware of, such as cross-site scripting attacks for HTTP and cross-site WebSocket hijacking for WebSocket.

However, if you configure them appropriately and use TLS encryption, you can mitigate most threats, making both protocols sufficiently secure.

Hybrid approaches to communication protocols

Typically, the recommended approach is to use both protocols for what they're best at within your system. That means using an HTTP connection for most of your standard web traffic and a WebSocket connection for anything that requires real-time communication, such as notifications, messaging/chat, or video calls.

You might also consider assessing complementary or alternative technologies; WebSocket and HTTP aren't the only options when it comes to real-time communication, after all. WebRTC is similar to WebSocket, with the key difference being that it's used to implement peer-to-peer connections without relying on a server. That can be especially helpful for video calls, allowing participants to communicate directly without introducing load to your server.

Communication protocols for secure, scalable, reliable in-app comms

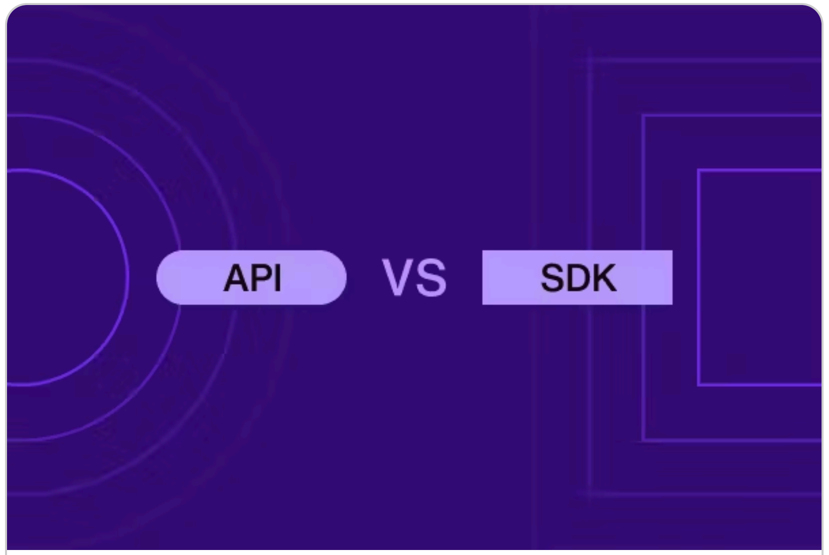
You should now have a solid idea now of how WebSocket and HTTP communication protocols are intended to be used. You've seen their strengths and weaknesses, and you can appreciate their tradeoffs.

Fortunately, you don't need to choose one over the other. The two communication protocols can, and often should, be used together, allowing each to do what it does best. If you need to implement real-time communication and streaming functionality in your applications, check out Sendbird. Sendbird's APIs and client SDKs handle the technical heavy lifting for in-app chat, calls, and live streaming. Sendbird's chat service abstracts away the problems (such as running reliably and securely in real-time at massive scale) associated with growing and maintaining a chat system over a long period of time. This means that you can focus on the core aspects of your application.

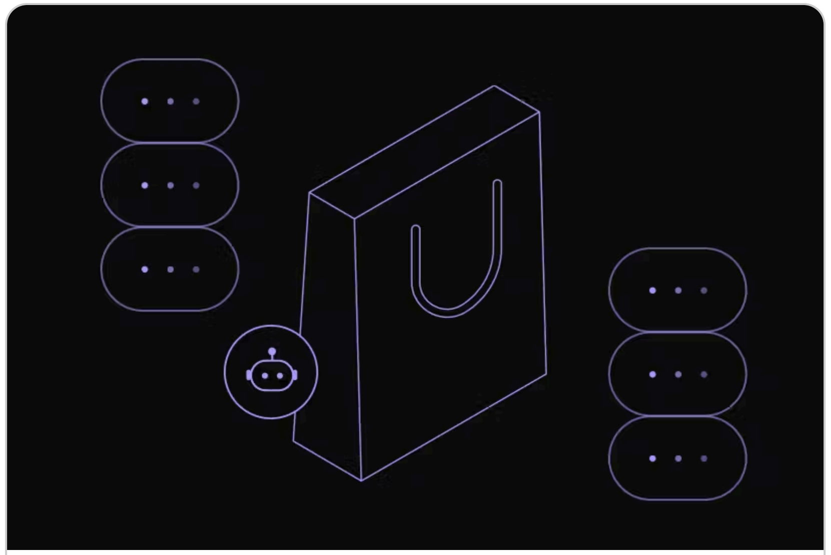
Want to try it out? Sign up for free—no commitment or credit card required.

Happy in-app communications building! 📱

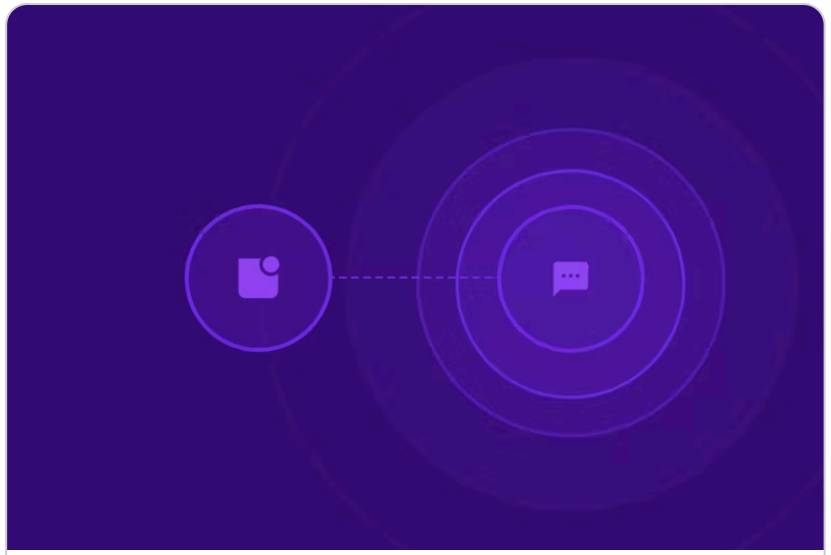
Related Tutorials



API vs. SDK: Differences with examples



How to build an AI chatbot for retail and ecommerce customers: A step-by-step guide



The step-by-step guide to messaging mobile customers using your app and SMS

01 / 02

Need more resources?

Sendbird Community

Sendbird docs

Enterprise AI customer service

→

AI SOLUTIONS

AI customer service
AI agent platform
AI agent builder

COMMUNICATION SOLUTIONS

Chat API
Content moderation
Salesforce connector
Support agent desk
Business messaging
Voice and video call API

PRICING

Pricing

RESOURCES

Customer stories
Ebooks & guides
Webinars

BLOG

What is an AI agent?
What is agentic AI?
How to build an AI agent
AI agent vs. chatbot
How to build an AI chatbot

DEVELOPER

Documentation
Demos
Tutorials
Learning center
Community
FAQ
Server status

CHAT SOLUTIONS BY INDUSTRY

On-demand
Retail
Financial services
Digital health
Marketplaces

CHAT SOLUTIONS BY USE CASE

Customer service
Sales
Marketing
Operations
Live streaming
User conversations

COMPANY

About
Customers
Careers
News
Partners
Security & compliance
RFP template

SUPPORT

Help center
Support policy
Contact sales

ACCOUNT

Log in
Free trial

Terms of service   Privacy notice   Your privacy choices   Sub-processors