



AVIOX MAGAZINE

LIVE CONNECTIONS,
SMARTER **AUTOMATION**
AND FASTER APPS

Real-Time and **Automation**

VOL. 1 • NO. 2
AUGUST 2026

In this issue

AI	3
Where RPA Meets AI: Designing Automation That Holds Up	3
RESEARCH	5
Measure Before You Optimize: A Profiling-Led Approach to React Performance	5
TECHNOLOGY	8
WebSockets and the Architecture of the Real-Time Web	8
Deploying FastAPI on AWS Lambda with Mangum	10

FROM THE EDITOR

This month is about speed: data that arrives the moment it changes, automation that keeps up, and apps that feel fast because someone measured them.

We look at where robotic process automation meets AI and how to design automation that holds up, and why you should profile a React app before trying to make it faster. Then we explain the architecture behind WebSockets and the real-time web, and show how to deploy FastAPI on AWS Lambda with Mangum.

As always, every story links back to its full version on Aviox Blogs.

— *The Aviox Blogs Editors*

© 2026 Aviox Blogs. All rights reserved, except where noted: stories republished from other publishers remain theirs and are credited with their licence. No other part of this publication may be copied, changed or republished without written permission from Aviox Blogs.
Read this issue online at <https://blogs.avioxtechnologies.com/magazine/real-time-and-automation>

AI

Where RPA Meets AI: Designing Automation That Holds Up

Bots that click through screens are brittle. Automation that understands documents, calls APIs and is monitored like any other system is not.

By Aviox Blogs Editorial · 12 Aug 2026 · 2 min read



Robotic process automation earned an uneven reputation. The promise was simple: software robots that do the repetitive work people shouldn't have to. The reality, too often, was bots that broke whenever a screen changed, a field moved or a document arrived in an unexpected format.

The automation that lasts looks different. It begins with understanding the process, uses AI where judgment is needed, prefers integrations to imitation, and is monitored like any other production system.

Understand the process before automating it

Automating a broken process only makes it fail faster. The first phase of any engagement is analysis: identifying manual tasks, mapping workflow gaps and finding the steps that are genuinely ready for automation. AI-assisted process discovery can surface patterns in how work actually flows — which is frequently different from how the documentation says it flows.

Automating a broken process only makes it fail faster.

Aviox Blogs Editorial

Design before you build

With the process mapped, automation design sets out workflows, decision logic and bot behavior on paper first. This is where exceptions are decided: what the bot does when data is missing, who is notified, and when a case returns to a person. Clarity here is what makes the eventual automation consistent rather than surprising.

This story continues for Dispatch subscribers — about 1 more minute of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

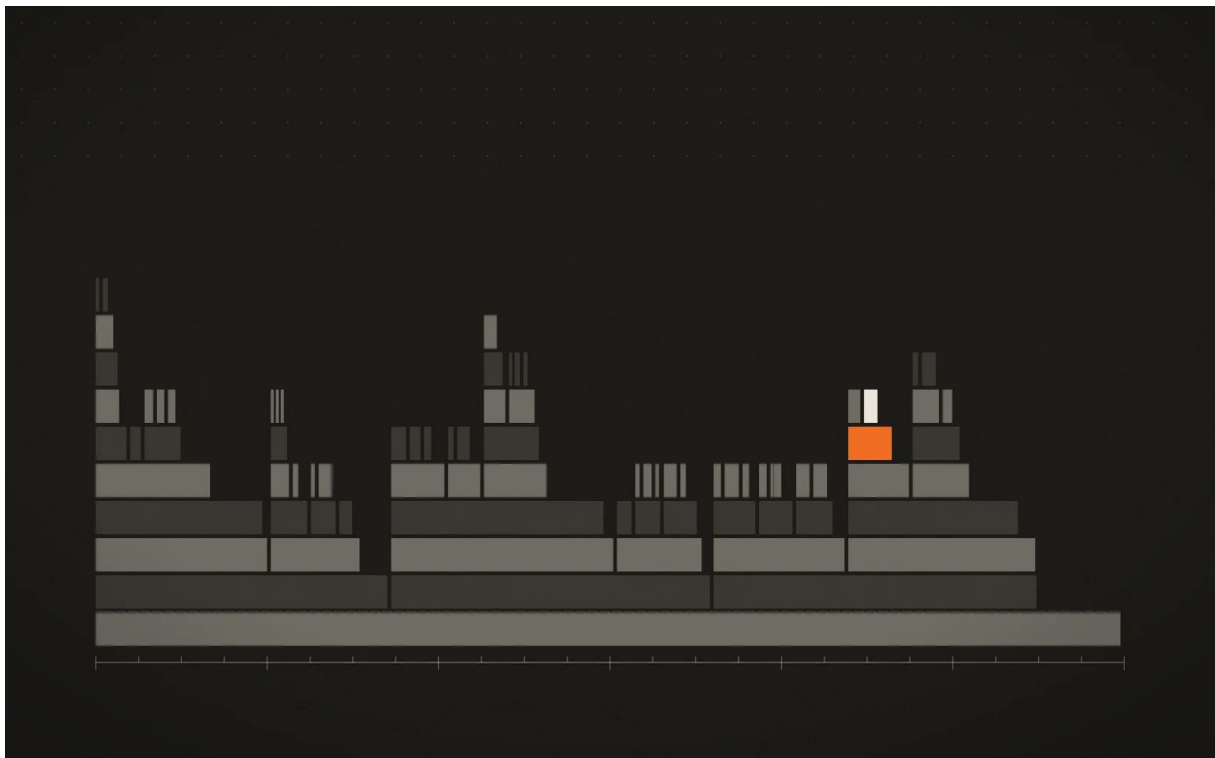
Read online: <https://blogs.avioxtechnologies.com/story/where-rpa-meets-ai-intelligent-automation>

RESEARCH

Measure Before You Optimize: A Profiling-Led Approach to React Performance

Memoization, code splitting and server rendering all have costs. This study argues for profiling first — and maps each common bottleneck to the technique that actually addresses it.

By Aviox Blogs Engineering · 5 Aug 2026 · 7 min read



Introduction

Before optimizing a React application, we must understand how React updates the interface and how to measure an application's performance. Optimization means reducing load time, minimizing memory use and keeping interactions responsive as an application grows in complexity and traffic. Done well, it improves user engagement, can reduce hosting costs, supports search ranking — page speed is a ranking factor — and often leaves code more modular and easier to maintain.

Done badly, it adds complexity in exactly the wrong places. The thesis of this paper is simple: measure first.

How React updates the interface

When state changes, React re-renders the affected components to produce a new element tree, compares it with the previous one, and applies only the differences to the DOM. Work therefore happens in two phases: a **render** phase, where components run and the new tree is computed, and a **commit** phase, where changes are written to the DOM.

Diagram: a state change triggers the render phase, where components re-run and the new tree is diffed against the previous one; the commit phase then applies only the changed nodes to the DOM.

Render is where components run; commit is where the DOM changes. Profilers measure both.

Common classes of bottleneck

Unnecessary re-renders

A state change high in the tree re-renders everything below it, including components whose output does not change.

Large component trees

Deeply nested trees are slow to render and update, and amplify every unnecessary re-render.

Inefficient data fetching

Waterfalls and over-fetching delay meaningful content and inflate load times.

Bloated bundles

Large JavaScript bundles take longer to download, parse and execute, delaying interactivity.

Memory leaks

Subscriptions and timers that outlive their components degrade performance until the app slows or crashes.

Improper state use

Setting state in loops or storing derived data multiplies renders without adding information.

Unoptimized images

Oversized images dominate transfer size and memory on many pages.

Method: profiling

Profiling measures where time is actually spent. The React DevTools browser extension includes a **Profiler** tab; Chrome DevTools' Performance panel and Lighthouse complement it for load and runtime analysis.

Capturing a profile

- 1 Open the React DevTools **Profiler** tab and press **Record**.
- 2 Use the application as a person would — the profiler records every commit.
- 3 Press **Record** again to stop and inspect the session.

The flame graph shows each commit as a tree whose bar widths represent render time; the ranked view sorts components by cost. Start with the widest bars and the components that render most often.

Measuring in code

For repeatable measurements — or to send timings to monitoring — wrap a subtree in the **Profiler** component. Its `onRender` callback fires after every commit of that subtree:

```
import { Profiler } from "react";

function onRender(id, phase, actualDuration, baseDuration) {
  // phase: "mount" | "update" | "nested-update"
  console.log(`\`${id}\` \`${phase}\`: \`${actualDuration.toFixed(1)}ms (base \`${baseDuration.toFixed(1)}ms\`)\`);
}
```

```
export default function App() {  
  return (  
    <Profiler id="Dashboard" onRender={onRender}>  
      <Dashboard />  
    </Profiler>  
  );  
}
```

`actualDuration` is the time spent rendering this commit; `baseDuration` estimates the cost of rendering the whole subtree without memoization. A large gap between them shows memoization is working; a small one shows it is not helping.

From data to change

- 1 **Collect** a profile of a realistic interaction.
- 2 **Analyze** it for the components that take the most time or render most often.
- 3 **Prioritize** the changes likely to have the largest effect.
- 4 **Optimize** narrowly — one bottleneck at a time.
- 5 **Verify** with a new profile, comparing against the original.
- 6 **Iterate** until the interaction meets its budget.

This story continues for Dispatch subscribers — about 4 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

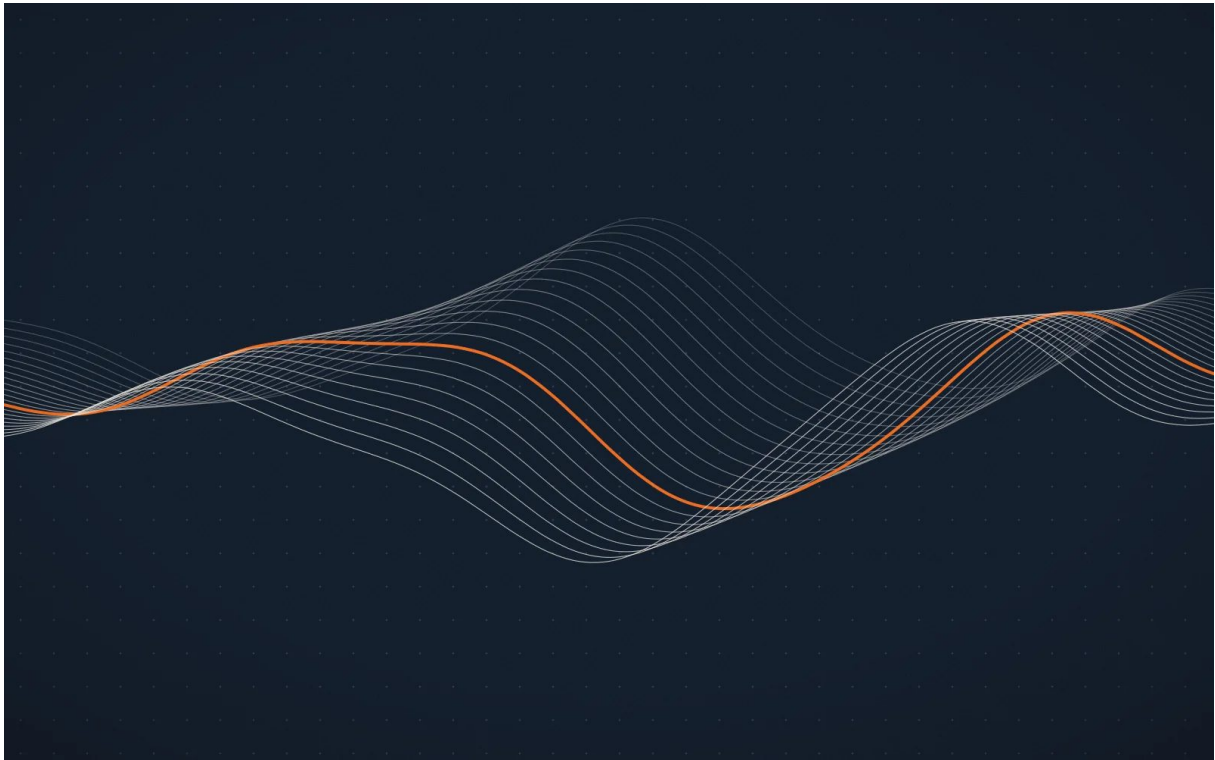
Read online: <https://blogs.avioxtechnologies.com/story/measure-before-you-optimize-react-performance>

TECHNOLOGY

WebSockets and the Architecture of the Real-Time Web

HTTP asks and waits. WebSockets keep a single connection open so the server can speak first — and that changes both what you can build and how you must secure it.

By Aviox Blogs Engineering · 27 Aug 2026 · 4 min read



WebSocket is a full-duplex, bidirectional protocol for communication between a client and a server. Its URLs begin with `ws://` or, when encrypted, `wss://`. Unlike HTTP, it is stateful: once the connection is established it stays open until one side closes it, and both parties can send messages at any moment over that same channel.

That single property — a connection that persists — is what makes live dashboards, multiplayer games and chat feel instantaneous. It is also what makes WebSockets behave very differently from the request-and-response model most web infrastructure was designed around.

How a WebSocket connection is made

Every WebSocket starts life as an ordinary HTTP request. The client asks the server to upgrade the connection; if the server agrees, the two complete a handshake and the underlying TCP connection is handed over to the WebSocket protocol. From then on, messages travel as lightweight frames in either direction until one side closes the channel.

Sequence diagram: the client sends an HTTP Upgrade request, the server replies 101 Switching Protocols, then messages flow in both directions over the open connection until a close frame ends it.

One HTTP upgrade, then a long-lived, two-way channel.

The browser API is deliberately small. A client opens a socket, listens for messages and sends its own whenever it needs to:

```
const socket = new WebSocket("wss://example.com/prices");

socket.addEventListener("open", () => {
  socket.send(JSON.stringify({ type: "subscribe", symbol: "AVX" }));
});

socket.addEventListener("message", (event) => {
  const update = JSON.parse(event.data); // treat server data as data
  renderPrice(update);
});

socket.addEventListener("close", (event) => {
  console.info("Socket closed", event.code, event.reason);
});
```

client.js

Where WebSockets earn their place

Real-time data

Trading platforms and live dashboards push price movements and metrics to the client the moment they change, over a connection that is already open.

Multiplayer and gaming

State arrives continuously and the interface updates without a refresh or a new connection for each change.

Chat and collaboration

A single connection handles sending, receiving, publishing and broadcasting messages between participants.

This story continues for Dispatch subscribers — about 3 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

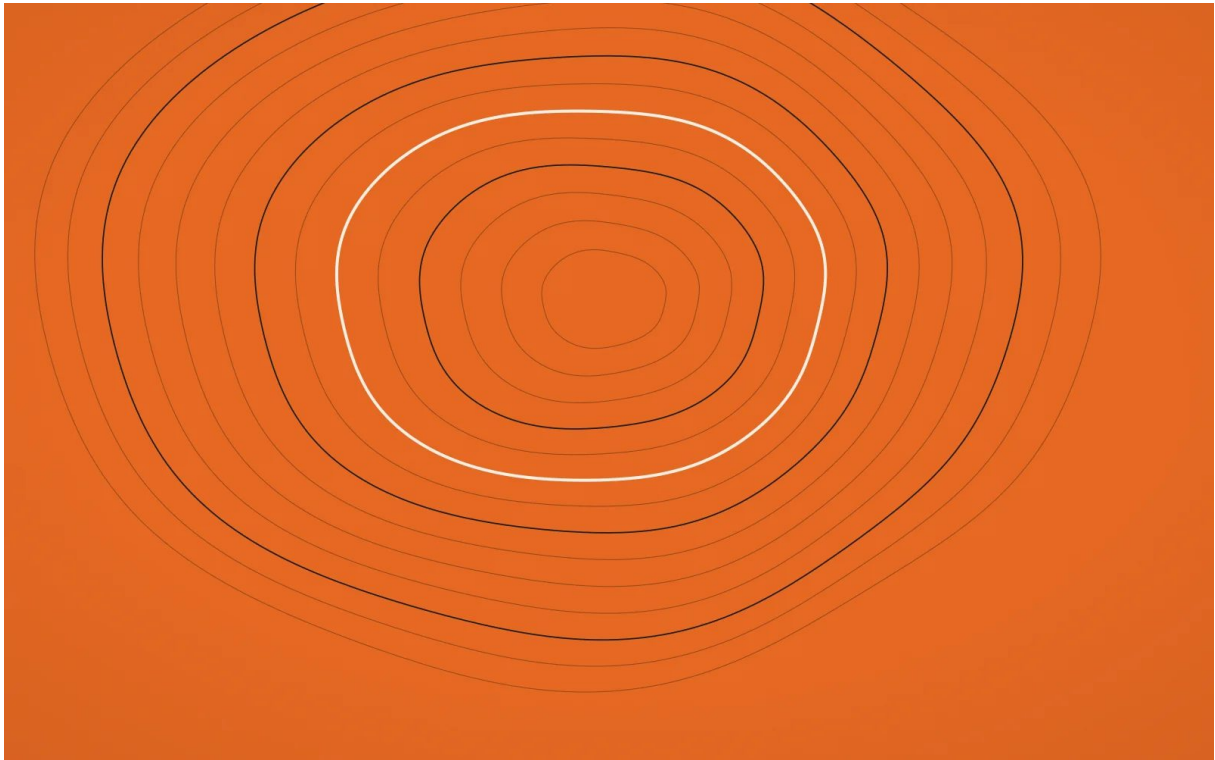
Read online: <https://blogs.avioxtechnologies.com/story/websockets-and-the-real-time-web>

TECHNOLOGY

Deploying FastAPI on AWS Lambda with Mangum

No servers to patch, nothing to pay for while idle. Package a FastAPI project as a Lambda function — code in one archive, dependencies in a reusable layer.

By Aviox Blogs Engineering · 20 Aug 2026 · 3 min read



AWS Lambda runs code on demand and bills only while it executes. FastAPI is an ASGI application, and Lambda speaks in events — so we need an adapter between the two. That adapter is Mangum.

This guide deploys a project using two separate archives: the application code, and its dependencies packaged as a Lambda layer. Keeping them apart means code-only changes upload in seconds, and the layer can be shared between functions.

Before you start

You'll need a FastAPI project and an AWS account with access to Lambda and S3. The examples use a project called `sample-fastapi-project`, an S3 bucket called `sample-fastapi-s3`, and a function called `sample-fastapi-lambda`.

Prepare the project

Isolate the project's dependencies in a virtual environment and install them:

```
python3 -m venv venv
source venv/bin/activate      # Windows: .\env\Scripts\activate
pip install -r requirements.txt
```

Add Mangum and record it in your requirements:

```
pip install mangum
pip freeze > requirements.txt
```

Then wrap the application. The `handler` object is what Lambda will invoke:

```
from fastapi import FastAPI
from mangum import Mangum

app = FastAPI()
```

```
@app.get("/health")
def health():
    return {"status": "ok"}
```

```
handler = Mangum(app)
```

```
main.py
```

This story continues for Dispatch subscribers — about 2 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

Read online: <https://blogs.avioxtechnologies.com/story/deploying-fastapi-on-aws-lambda>