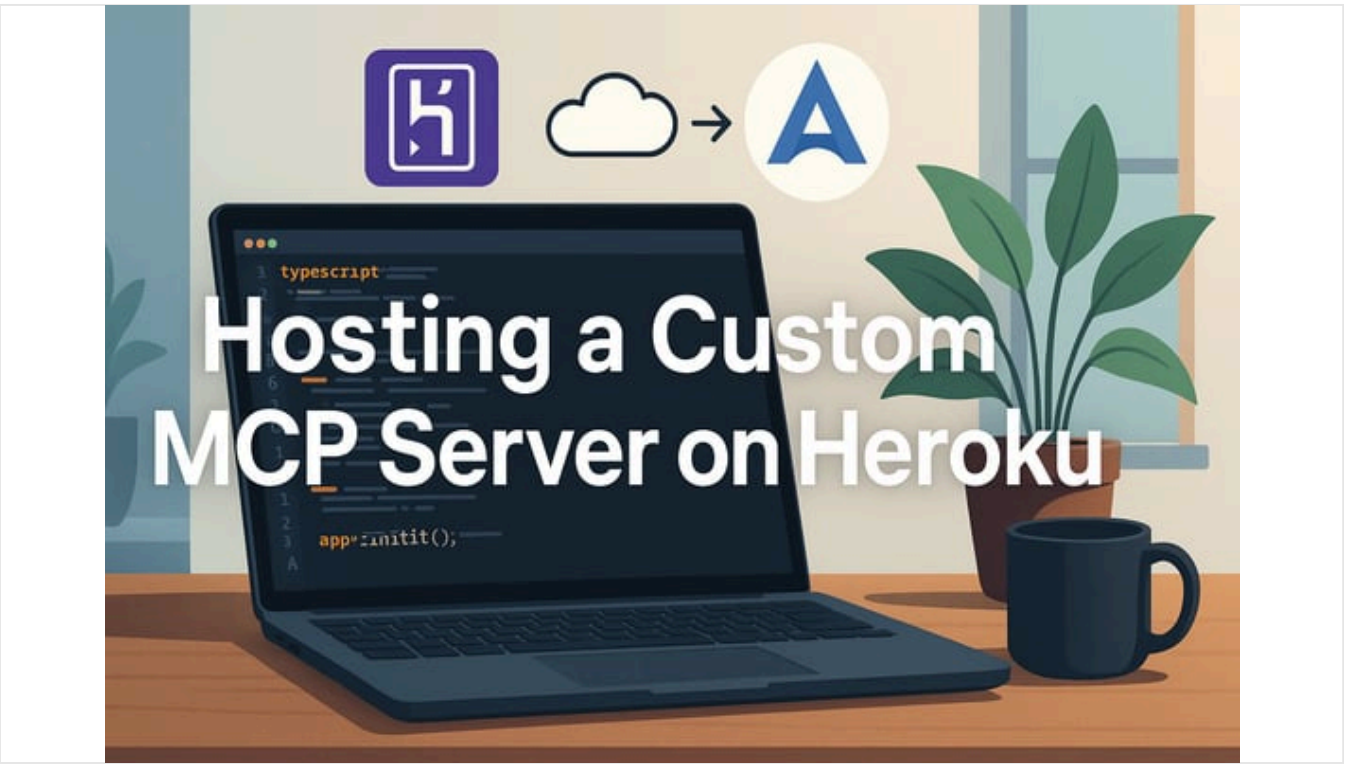
Building and Deploying a TypeScript MCP Server on Heroku

Published July 31, 2025 20 min read



Hosting a Custom MCP Server on Heroku in 30 Minutes

Author's Note: This guide walks through creating and deploying a Model Context Protocol (MCP) server with TypeScript and Node.js, and integrating it with Salesforce Agentforce via **Heroku AppLink**. We'll cover project setup, development vs. production configurations, implementing basic MCP handlers (tools), deploying to Heroku, and ensuring enterprise-grade security, scaling, and troubleshooting. The goal is a professional, production-ready deployment – all in about 30 minutes of work.

1. Project Scaffolding: TypeScript MCP Server Setup

Before coding, scaffold your Node.js/TypeScript project and include the MCP SDK and other dependencies:

- **Initialize the Node project:** Create a project directory and run `npm init -y` to generate a basic `package.json` (Source: freecodecamp.org). Ensure your `package.json` has `"type": "module"` (so Node can use ES module imports).
- **Install MCP SDK and basics:** Add the official MCP TypeScript SDK and related libraries. For example:

  ```
  npm install @modelcontextprotocol/sdk express zod dotenv
  ```

  This installs the MCP server/client SDK (Source: freecodecamp.org), Express (for HTTP server), Zod (for input schema validation), and dotenv (for local environment config).
- **Project structure:** For simplicity, start with a single file server (e.g. `src/index.ts`). In our example, **all code lives in one file** for brevity (Source: freecodecamp.org), but you can organize into multiple modules as needed (e.g. a `tools/` directory for tool handlers, etc.). Create a `tsconfig.json` to compile to ES2019 (or later) and output to a `dist/` folder for production.
- **TypeScript configuration:** Ensure TypeScript is set up for Node. For example, a minimal `tsconfig.json` might include:

  ```
  {
    "compilerOptions": {
      "target": "ES2020",
      "module": "ESNext",
      "moduleResolution": "Node",
      "outDir": "dist",
      "esModuleInterop": true
    },
    "include": ["src"]
  }
  ```

  This compiles modern JS, uses ES modules, and outputs build files to `dist/`.

- **Environment variables:** Create a `.env` file for sensitive configs (like API keys) during development (Source: freecodecamp.org). For example, if your MCP server will call external APIs (Google, etc.), put keys in `.env` and load them via `dotenv` in development. (We'll configure these vars on Heroku in production.)

By the end of this step, you have a Node.js project with TypeScript ready, and the MCP SDK installed. Next, we'll implement the MCP server code.

2. Development vs. Production Environment Setup

During development, you can run the TypeScript server directly, but in production we'll compile to JavaScript for efficiency:

- **Development workflow:** Use `ts-node` or a build watcher for quick iteration. For example, add a **npm script** for development: `"dev": "ts-node src/index.ts"`. You can use `nodemon` for auto-reload during development. This lets you run the TS code without a manual compile step.
- **Production build:** It's **not recommended** to use `ts-node` in production (Source: stackoverflow.com)(Source: stackoverflow.com). Instead, compile your TypeScript to JavaScript. Add a **build script** (e.g. `"build": "tsc"`) and a **start script** to run the compiled code (e.g. `"start": "node dist/index.js"`). After development, run `npm run build` to produce `dist/index.js` and use `npm start` to launch. This avoids on-the-fly transpilation overhead and aligns with Heroku best practices (Source: stackoverflow.com).
- **Scripts and dependencies:** Make sure any necessary runtime dependencies (like `ts-node` or TypeScript) are properly classified. If you choose to run TS directly in Heroku (quick and dirty approach), include `ts-node` in the **dependencies** (not just devDependencies) and set the start script accordingly (Source: stackoverflow.com). However, the **recommended approach** is to compile first – yielding faster startup and execution.

Setting up these scripts prepares your app for both local iteration and Heroku deployment. Now, let's write the actual MCP server code and define a basic tool/handler.

3. Implementing Basic MCP Protocol Handlers (Tools)

In MCP, *tools* are the functions (actions) your AI agent can invoke on the server (Source: freecodecamp.org). We'll create a simple MCP server with one or two tools to illustrate the pattern:

- **Import and instantiate MCP server:** In `src/index.ts`, import the `McpServer` from the SDK. For example:

```
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StreamableHTTPServerTransport } from "@modelcontextprotocol/sdk/server/stre
import { z } from "zod";
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
dotenv.config();
const server = new McpServer({ name: "My Custom MCP", version: "1.0.0" });
```

  The `McpServer` is the core of your MCP implementation – it manages connections, protocol compliance, and routing of messages (Source: github.com). We name the server and set a semantic version.

- **Define MCP tools:** Use `server.tool(name, schema, handler)` to register functions. For example, let's add a simple tool that echoes a greeting, and one that adds two numbers:

  typescript

  Copy

```
server.tool( "greetUser", { name: z.string() }, // input validation schema async ({
name }) => { return { content: [{ type: "text", text: `Hello, ${name}!` }] }; } );
server.tool( "addNumbers", { a: z.number(), b: z.number() }, async ({ a, b }) => {
const sum = a + b; return { content: [{ type: "text", text: `Sum is ${sum}` }] }; }
);
```

  Explanation: We give each tool a meaningful name (e.g. `"greetUser"`), define input parameters with Zod schemas, and provide an async handler function (Source: freecodecamp.org). The handler returns a response object following MCP's structured format (here simply returning text content). In a real scenario, tools might fetch data from an API or database. For instance, an MCP server could integrate with Google Calendar (Source: freecodecamp.org)(Source: freecodecamp.org) or a database to retrieve info, and return it as structured content for the AI to consume.

- **Initialize server transport (HTTP):** To allow clients (e.g. Agentforce or an AI IDE) to connect, our MCP server must listen on an endpoint. We use Express and the MCP **Streamable HTTP** transport for a web-accessible server. For example:

```
const app = express();
app.use(express.json()); // Enable CORS for specific origins or all, exposing the MC
```

In the above setup, we create an Express app and a single `/mcp` route that handles MCP JSON-RPC requests (POST for requests, GET for SSE stream, DELETE to close) (Source: github.com)(Source: github.com). We maintain a map of transports for active sessions. When an AI client first connects (sends an `initialize` call), we create a new `StreamableHTTPServerTransport`, attach our `McpServer` to it, and store it. Subsequent requests with the same `Mcp-Session-Id` header are routed to the existing session transport (Source: github.com)(Source: github.com). This allows the client (e.g. Agentforce or Claude) to maintain a continuous session with our server over HTTP/SSE. We also enable CORS and explicitly expose the `Mcp-Session-Id` header, which is **required for browser-based MCP clients** to work properly (Source: github.com)(Source: github.com) (Agentforce uses server-to-server calls, so CORS is less a concern there, but during testing with web UIs it matters).

- **Port and server startup:** Finally, start the Express server:

  typescript

  Copy

  ```
  const PORT = process.env.PORT || 3000; app.listen(PORT, () => { console.log(`✅ MCP server listening on port ${PORT}`); });
  ```

  We bind to the port from environment. Heroku will supply a port via the `PORT` env var at runtime, which your app **must** use (Source: devcenter.heroku.com). (Locally, we default to 3000 for convenience.)

At this point, we have a functional MCP server with basic tools. You can test it locally using an MCP client (for example, Anthropic's Claude or the MCP Inspector tool). For instance, using the open-source **MCP Inspector** utility is helpful: run `npx @modelcontextprotocol/inspector`, open the UI, and connect to `http://localhost:3000/mcp` with transport type "HTTP" or "SSE" to invoke your tools. Once satisfied, we'll prepare to deploy this to Heroku.

4. Deploying the MCP Server to Heroku (CLI, Dynos, Env Vars, Logs)

Deploying a Node.js app to Heroku is straightforward. Let's assume you have a Heroku account and the Heroku CLI installed:

- **Login and create app:** Log in with `heroku login`. Then from your project directory, create a Heroku app:

  ```
  heroku create
  ```

  This creates an app (with a random name if not specified) and a Git remote URL (Source: devcenter.heroku.com). For example, you might see: *Creating app... done,* ⬢ `my-mcp-app` *https://my-mcp-app.herokuapp.com/*.

- **Set buildpack for Node.js:** Heroku auto-detects Node.js via the `package.json`. Ensure your repo is a git repo and commit your code. If you used the standard Node buildpack, no manual action is needed (it's automatic on push). If you plan to use Heroku AppLink's service mesh (discussed in the next section), you will add an extra buildpack, but for a basic deployment we start simple.

- **Configure environment variables:** Any secrets or config (like API keys, etc. that you had in `.env`) should be set in Heroku's config vars. Run `heroku config:set KEY=value` for each variable (Source: devcenter.heroku.com). For example, to set a Google API key:

  ```
  heroku config:set GOOGLE_PUBLIC_API_KEY=your_key_here heroku config:set CALENDAR_ID=
  ```

  These will be available as `process.env.GOOGLE_PUBLIC_API_KEY` in your app.

- **Procfile (dyno configuration):** Create a file named **`Procfile`** in the project root to declare how to run the app. Since our app is a web server, define a `web` process. For example:

  arduino

  Copy

  ```
  web: npm run start
  ```

This tells Heroku to execute `npm run start` (which runs our compiled `dist/index.js`) in a web dyno. It's important the process listens on `$PORT` (which we handled in code). The Procfile isn't strictly required if you have a start script, but it's good practice to be explicit (Source: stackoverflow.com). If using the AppLink service mesh buildpack later, the Procfile may include a special command (more on that shortly).

- **Deploy via Git:** Push the code to Heroku's Git remote:

  ```
  git add . git commit -m "Deploy MCP server" git push heroku main
  ```

  Heroku will build the app (running `npm install`, `npm run build` if you specified a build step, etc.) and release it. On success, you can run `heroku ps` to see the dyno running and `heroku open` to open the app URL. (Our MCP server doesn't serve a web page, but the endpoint `/mcp` is ready for agent connections.)

- **Logging and monitoring:** Use `heroku logs --tail` to stream logs from the app in real-time (Source: devcenter.heroku.com). This is crucial for debugging issues (e.g., if the app crashes on startup or when a request comes in). Heroku's logging will show console output, including our startup message and any errors or `console.log` from tool handlers. For deeper debugging, you can also run `heroku run bash` to get a shell in the dyno for inspection.

At this stage, your custom MCP server is live on Heroku, accessible at `https://<your-app>.herokuapp.com/mcp`. You could connect an AI client (like Claude or a custom MCP client) to this URL (it supports **HTTP+SSE** MCP protocol). Next, we'll integrate this with **Agentforce** using Heroku AppLink for enterprise governance and security.

5. Integrating the MCP Server with Agentforce via Heroku AppLink

Salesforce's **Agentforce** allows enterprise AI agents to perform actions securely. With **Agentforce 3.0**, Salesforce introduced native MCP support, and Heroku's **AppLink** feature lets you connect external services (like your MCP server) to Agentforce in a governed, secure way (Source: salesforceben.com). Here's how to integrate our deployed MCP server:

- **Heroku AppLink overview:** Heroku AppLink is an add-on that exposes Heroku apps as API services inside Salesforce (including Agentforce) (Source: devcenter.heroku.com)(Source: devcenter.heroku.com). When you attach AppLink to your app, it handles authentication,

authorization, and connectivity between Salesforce and your Heroku app. In practice, AppLink will ensure that only *trusted Salesforce orgs* (that you connect) can call your MCP endpoints, and it can auto-generate **Agentforce Actions** from your API definitions.

- **Provision AppLink on your app:** Use the Heroku CLI to add the AppLink add-on to your Heroku app. For example:

  ```
  heroku addons:create heroku-applink
  ```

  (Ensure you meet any requirements, e.g. certain dyno types or a verified Heroku account (Source: devcenter.heroku.com).) This attaches the AppLink service to your app. You should also install the Heroku AppLink CLI plugin: `heroku plugins:install @heroku-cli/plugin-applink` (Source: devcenter.heroku.com), which gives you commands to connect to Salesforce.

- **Attach Salesforce org and publish API:** You'll need a Salesforce org with Agentforce enabled (a Developer Edition or sandbox with Agentforce) (Source: devcenter.heroku.com). Using the AppLink CLI, connect your Heroku app to Salesforce. For example:

  ```
  heroku salesforce:connect -a your-heroku-app # connects a Salesforce org (you'll loc
  ```

  The `connect` step authorizes a Salesforce org to your Heroku app (establishing the trust). The `publish` step registers your app's API with Salesforce, making it discoverable. Under the hood, this uses an **OpenAPI specification** file in your app (if present) to define the actions.

- **Provide an OpenAPI spec for your MCP endpoints:** To integrate nicely, you should describe your MCP server's API in an OpenAPI 3.0 spec (YAML/JSON). This spec acts as a contract that Salesforce reads to generate **External Services** and Agentforce **Custom Actions**(Source: heroku.com). In our case, the API is basically one endpoint (`/mcp`) that accepts a generic MCP request. However, the more meaningful integration is to define *specific actions/tools* in the spec. For example, you could define an endpoint for each tool (like `/mcp/tools/greetUser` or similar) in the spec purely for documentation/integration purposes. Salesforce's AppLink uses the OpenAPI and special extensions (`x-sfdc` fields) to automatically create corresponding Agentforce actions and map Salesforce permission sets to them (Source: heroku.com).

  In the **Getting Started with Heroku AppLink and Agentforce** guide, a sample `api-spec.yaml` is provided (Source: devcenter.heroku.com). Adapting that to our scenario, you'd list operations corresponding to your MCP tools, and mark them as Agentforce actions. Once this spec is in your repo (and committed), running `heroku salesforce:publish` will register those endpoints. Salesforce administrators can then see these as available actions.

- **Apply the AppLink buildpack (service mesh): Crucial for security**, the AppLink Service Mesh buildpack must be added to your app **before** the Node buildpack. This injects a proxy in front of your app to enforce Salesforce auth. Add it by running:

  ```
  heroku buildpacks:add --index 1 heroku/applink-service-mesh
  ```

  Then redeploy (`git push heroku main` again). The service mesh will require that incoming requests present valid Salesforce-signed JWT tokens and originate from your connected org. It **blocks any external access** not coming through Salesforce channels (Source: heroku.com). Essentially, after this, your Heroku app will *only* serve requests from Agentforce (or other Salesforce flows you connected), preventing direct abuse of your `/mcp` endpoint. This addresses enterprise governance: you get fine-grained control via Salesforce over who/what can invoke the tools.

- **Enterprise governance via Agentforce:** Once published, your MCP server's tools become **Agentforce Actions** within your Salesforce org. Admins can assign these actions to Agentforce agents, include them in Agentforce policies, and restrict their usage with Salesforce's permission sets. Agentforce's governed gateway will list your custom tool actions and allow enabling or disabling them per agent (Source: salesforceben.com). All calls from Agentforce carry the user context and adhere to the mode (User mode, etc.) configured in AppLink (Source: devcenter.heroku.com). For example, if AppLink is in *user-plus mode*, the action runs with the invoking user's permissions plus optional elevated rights defined by you (Source: devcenter.heroku.com). This ensures any data access or changes done by your MCP tool comply with Salesforce security and role policies.

- **Testing the integration:** In Salesforce's Agentforce interface, you should now see the custom action corresponding to your MCP server (if configured via OpenAPI). For instance, "greetUser" might appear as an action that can be invoked by an agent. You can create an Agentforce agent (via Prompt Builder or Agent Builder) that uses this action. When triggered, Agentforce will call your Heroku MCP server (through AppLink's secure channel), your server will execute the tool, and the result will return to the agent. Monitor your Heroku logs during a test run to see the interaction (the logs should show an incoming POST to `/mcp` with the JSON request).

Using Heroku AppLink in this way gives you **the best of both worlds**: the flexibility of a custom Node/TypeScript service and the governance and scalability of the Salesforce platform. Salesforce's documentation emphasizes that developers can "spin up and expose custom MCP services using Heroku, offering a fast, secure, and scalable way to connect bespoke tools to Agentforce" (Source: salesforceben.com).

6. Security, Scaling, and Troubleshooting Considerations

Finally, let's address some important operational aspects:

- **Security best practices:** By default, our MCP server has no authentication on its own. If you deploy it without AppLink, **anyone with the URL could attempt to use it**, so you'd want to implement auth (e.g. an API key or allowlist). However, using **Heroku AppLink's service mesh** mitigates this by only allowing verified Salesforce traffic (Source: heroku.com). The service mesh handles OAuth and token verification, so you don't need to add custom auth checks for Agentforce – those are handled for you. Still, you should treat your Heroku app like any production service: keep dependencies updated, handle errors to avoid leaking info, and use HTTPS (Heroku provides SSL by default on *.herokuapp.com domains). If your MCP server calls external APIs, **never hard-code secrets** – use config vars and the Heroku secure config store. Also consider enabling Heroku's web application firewall or IP restrictions if necessary (though AppLink's mesh will already restrict traffic drastically).

- **Environment and configuration management:** In enterprise settings, you might have multiple Salesforce orgs (dev, test, prod) and corresponding Heroku apps. Heroku AppLink supports connecting multiple orgs and even multiple Heroku apps to the same add-on for flexibility (Source: devcenter.heroku.com)(Source: devcenter.heroku.com). Manage your config vars per environment (e.g., use Heroku Pipelines with staging/production apps and config variables, or the CLI to set vars for each app).

- **Scaling and performance:** Our MCP server can be scaled horizontally on Heroku by increasing the number of web dynos (`heroku ps:scale web=2` for two dynos, etc.). Thanks to Heroku's stateless process model, adding more dynos can handle more concurrent Agentforce requests easily (Source: devcenter.heroku.com)(Source: devcenter.heroku.com). However, note that if you use the **stateful session approach** (like we did with session IDs stored in-memory), a given session will "stick" to the dyno that created it. In a multi-dyno scenario, Agentforce's subsequent requests might hit a different dyno that doesn't know the session. To avoid this, you could use a shared session store or implement the **stateless mode** from the MCP SDK (where each request is independent) (Source: github.com) (Source: github.com). Stateless mode may be simpler for scaling at the cost of re-initializing the server for each request (which, depending on your server's startup time, might be acceptable). In practice, Agentforce calls to tools are short-lived, so stateless operation can work well behind the scenes – Heroku can even spin up one-off dynos per request if using their Managed Inference API. For most use-cases, a single standard dyno can handle quite a few requests (MCP JSON exchanges are lightweight), so scale out as needed after monitoring.

- **Logging and monitoring:** We already mentioned `heroku logs`. For more advanced monitoring, consider add-ons like Papertrail or Datadog for log aggregation. Additionally, the **Heroku AppLink Dashboard** (accessible via `heroku addons:open heroku-applink`) provides a UI to monitor

connections, authorizations, and publications of your AppLink integration (Source: heroku.com). Salesforce's Agentforce Command Center will also show metrics on agent action usage and performance. Use these tools to monitor that your MCP server is being invoked as expected and to troubleshoot any errors in execution. If something goes wrong on the Salesforce side (e.g., a permission issue), the Agentforce logs or Flow errors will indicate it. Common issues include missing permission sets (which can be solved by adjusting the user mode or permission mapping in the OpenAPI spec), or forgetting to add the buildpack (leading to unauthorized access attempts).

- **Common errors and solutions:** If your Heroku app isn't responding, ensure it bound to the correct PORT (remember, Heroku sets `PORT` env var at runtime; using a hardcoded port will cause the app to crash on start) (Source: devcenter.heroku.com). If you see an `Error: address already in use` or similar, you might have hardcoded a port or started multiple servers. If Agentforce calls aren't reaching your app, check that the app is published and the Salesforce org is connected (run `heroku applink:connections -a your-app` to list connected orgs). Also confirm the Agentforce agent has the action enabled and that the buildpack was added (without it, external calls might be blocked or not authenticated). Heroku's AppLink logs can be viewed with the same `heroku logs` command – AppLink will log authentication events or errors there (Source: devcenter.heroku.com). The Heroku Dev Center provides a "AppLink Logging and Common Errors" document with specific troubleshooting steps for issues like missing JWT, incorrect OpenAPI spec, etc. (Source: devcenter.heroku.com).

- **Future considerations:** As the MCP and Agentforce ecosystem evolves, keep your MCP server updated. The MCP SDK is under active development (e.g., new protocol features or security patches) – track the official repository (Source: github.com). Additionally, Salesforce might introduce new integration features; for instance, **MuleSoft** can convert APIs into MCP services automatically (Source: salesforceben.com), which could complement your custom server. Always follow Salesforce's security review guidelines if your MCP server will be offered through the **AgentExchange** marketplace in the future.

*Figure: High-level MCP architecture – an AI **Host** (e.g. Agentforce or an AI IDE) connects via MCP to a custom **MCP Server** (your Heroku app). The MCP server interfaces with external data sources (APIs, databases, etc.) and returns structured data (context) to the host (Source: freecodecamp.org). In our deployment, Agentforce acts as the host with an integrated MCP client, calling our server through a secure channel.*

In summary, **hosting a custom MCP server on Heroku** combines familiar Node.js development with cutting-edge AI integration. We scaffolded a TypeScript project, wrote MCP handlers ("tools"), and set up a Node/Express server to handle MCP requests. We then deployed it on Heroku within minutes, using standard CLI workflows (Source: github.com). By leveraging **Heroku AppLink**, we integrated the service into Salesforce's Agentforce with enterprise-level security – Heroku acts as the bridge, enforcing OAuth

and org-level permissions (Source: heroku.com)(Source: devcenter.heroku.com). With this setup, professional developers can deliver custom AI tool integrations rapidly, while satisfying their organization's governance requirements. Happy coding, and may your MCP servers empower your AI agents safely and effectively!

Sources:

- Sumit Saha, *"How to Build a Custom MCP Server with TypeScript — A Handbook for Developers,"* freeCodeCamp (June 25, 2025) (Source: freecodecamp.org)(Source: freecodecamp.org) (Source: github.com).
- Model Context Protocol TypeScript SDK – GitHub README (Source: github.com)(Source: github.com) (Source: github.com).
- Heroku Dev Center, *"Getting Started with Heroku AppLink and Agentforce"* (July 2025) (Source: devcenter.heroku.com)(Source: devcenter.heroku.com); *"Heroku AppLink"* article (Source: devcenter.heroku.com)(Source: devcenter.heroku.com).
- Salesforce Ben, *"Salesforce Announces Agentforce 3.0 — Command Center, MCP, and Apps"* (June 23, 2025) (Source: salesforceben.com)(Source: salesforceben.com).
- Heroku Blog, *"Heroku AppLink: Extend Salesforce with Any Programming Language"* by Vivek Viswanathan & Kim Harrison (July 17, 2025) (Source: heroku.com)(Source: heroku.com).
- Heroku Dev Center, *"Runtime Principles"* (Dec 03, 2024) – on binding to PORT (Source: devcenter.heroku.com).
- Stack Overflow – discussions on deploying TypeScript apps to Heroku (on using ts-node vs. compiling) (Source: stackoverflow.com)(Source: stackoverflow.com).
- Heroku Dev Center – *"Configuration and Config Vars"* (retrieved 2025) (Source: devcenter.heroku.com) and Heroku CLI documentation.
- Heroku Dev Center – *"Working with MCP on Heroku"* (2025) (Source: devcenter.heroku.com). (For advanced use with Heroku's Managed Inference).

Tags: typescript, node.js, heroku, deployment, mcp, model context protocol, server development, salesforce

---

About Cirra

About Cirra AI

Cirra AI is a specialist software company dedicated to reinventing Salesforce administration and delivery through autonomous, domain-specific AI agents. From its headquarters in the heart of Silicon Valley, the team has built the **Cirra Change Agent** platform—an intelligent copilot that plans, executes, and documents multi-step Salesforce configuration tasks from a single plain-language prompt. The product combines a large-language-model reasoning core with deep Salesforce-metadata intelligence, giving revenue-operations and consulting teams the ability to implement high-impact changes in minutes instead of days while maintaining full governance and audit trails.

Cirra AI's mission is to **"let humans focus on design and strategy while software handles the clicks."** To achieve that, the company develops a family of agentic services that slot into every phase of the change-management lifecycle:

- **Requirements capture & solution design** – a conversational assistant that translates business requirements into technically valid design blueprints.
- **Automated configuration & deployment** – the Change Agent executes the blueprint across sandboxes and production, generating test data and rollback plans along the way.
- **Continuous compliance & optimisation** – built-in scanners surface unused fields, mis-configured sharing models, and technical-debt hot-spots, with one-click remediation suggestions.
- **Partner enablement programme** – a lightweight SDK and revenue-share model that lets Salesforce SIs embed Cirra agents inside their own delivery toolchains.

This agent-driven approach addresses three chronic pain points in the Salesforce ecosystem: (1) the high cost of manual administration, (2) the backlog created by scarce expert capacity, and (3) the operational risk of unscripted, undocumented changes. Early adopter studies show time-on-task reductions of 70-90 percent for routine configuration work and a measurable drop in post-deployment defects.

Leadership

Cirra AI was co-founded in 2024 by **Jelle van Geuns**, a Dutch-born engineer, serial entrepreneur, and 10-year Salesforce-ecosystem veteran. Before Cirra, Jelle bootstrapped **Decisions on Demand**, an AppExchange ISV whose rules-based lead-routing engine is used by multiple Fortune 500 companies. Under his stewardship the firm reached seven-figure ARR without external funding, demonstrating a knack for pairing deep technical innovation with pragmatic go-to-market execution.

Jelle began his career at ILOG (later IBM), where he managed global solution-delivery teams and honed his expertise in enterprise optimisation and AI-driven decisioning. He holds an M.Sc. in Computer Science from Delft University of Technology and has lectured widely on low-code automation, AI safety, and DevOps for SaaS platforms. A frequent podcast guest and conference speaker, he is recognised for advocating "human-in-the-loop autonomy"—the principle that AI should accelerate experts, not replace them.

Why Cirra AI matters

- **Deep vertical focus** – Unlike horizontal GPT plug-ins, Cirra's models are fine-tuned on billions of anonymised metadata relationships and declarative patterns unique to Salesforce. The result is context-

aware guidance that respects org-specific constraints, naming conventions, and compliance rules out-of-the-box.

- **Enterprise-grade architecture** – The platform is built on a zero-trust design, with isolated execution sandboxes, encrypted transient memory, and SOC 2-compliant audit logging—a critical requirement for regulated industries adopting generative AI.
- **Partner-centric ecosystem** – Consulting firms leverage Cirra to scale senior architect expertise across junior delivery teams, unlocking new fixed-fee service lines without increasing headcount.
- **Road-map acceleration** – By eliminating up to 80 percent of clickwork, customers can redirect scarce admin capacity toward strategic initiatives such as Revenue Cloud migrations, CPQ refactors, or data-model rationalisation.

Future outlook

Cirra AI continues to expand its agent portfolio with domain packs for Industries Cloud, Flow Orchestration, and MuleSoft automation, while an open API (beta) will let ISVs invoke the same reasoning engine inside custom UX extensions. Strategic partnerships with leading SIs, tooling vendors, and academic AI-safety labs position the company to become the de-facto orchestration layer for safe, large-scale change management across the Salesforce universe. By combining rigorous engineering, relentlessly customer-centric design, and a clear ethical stance on AI governance, Cirra AI is charting a pragmatic path toward an autonomous yet accountable future for enterprise SaaS operations.
