ChatGPT

Building an AI-Native Collaborative Tool with Convex (JavaScript Edition)

Developing a full-stack, AI-powered collaborative app can be greatly simplified using **Convex** – a reactive backend platform – combined with familiar JavaScript libraries on the frontend. In this guide, we walk through building a lightweight yet highly scalable architecture for a tool that integrates a chatbot, real-time analytics, and a CRM module. We'll focus on using Convex with **pure JavaScript** (no TypeScript required), leveraging Convex's **Chef** AI app builder for rapid development, and adhering to best practices for real-time features and scalability. We'll also delve into Convex's pricing model (usage-based) and compare it to platform pricing like Notion's (seat-based) to understand cost drivers.

Setting Up Convex with a Pure JavaScript Stack

Convex is designed to work seamlessly with TypeScript, but you can also use it with plain JavaScript. Since TypeScript is a superset of JavaScript, the Convex CLI and runtime will accept your JS code – you just won't get static type checking. To start, install Convex in your project and initialize a Convex backend:

```
npm install convex    # or pnpm add convex
npx convex dev        # log in (e.g. via GitHub) and init a Convex project
```

Running `npx convex dev` will prompt you to authenticate and then provision a Convex project. The CLI creates a `convex/` directory in your project and populates environment variables (like your deployment URL) in a local `.env` file [1]. For example, you'll see variables such as `CONVEX_DEPLOYMENT="dev:..."` and `NEXT_PUBLIC_CONVEX_URL="https://<your-app>.convex.cloud"` which the Convex client uses to connect. The `convex/` directory is where you'll write all backend logic (queries, mutations, actions) in JavaScript files. Convex will live-update your cloud backend as you modify these files, similar to how a frontend dev server updates on file saves [2].

Using Convex without TypeScript: By default, Convex generates TypeScript definitions for your backend functions (under `convex/_generated/`). In a pure JS setup, you can ignore or remove those, and instead use the **untyped API** to call functions. Convex provides an `anyApi` object for this purpose. For example, to call a Convex function `listMessages` defined in your backend, you can do:

```
import { ConvexClient } from "convex/browser";
import { anyApi } from "convex/server";

const client = new ConvexClient(process.env.CONVEX_URL);
client.onUpdate(anyApi.messages.list, {}, (msgs) => {
  console.log("Updated messages:", msgs);
});
await client.mutation(anyApi.messages.send, { body: "Hello!" });
```

Here we import `anyApi` and reference our functions by name (e.g. `messages.list` corresponds to `export const list` in `convex/messages.js`). This approach works entirely in JavaScript. Convex's JS client uses **WebSockets** under the hood to subscribe to query results and keep the UI in sync [3] [4], so the above `onUpdate` will continuously stream updates whenever the data changes. Meanwhile, `client.mutation` triggers a server-side mutation function via an HTTP call. In summary, with just JavaScript you can still get type-safe-like behavior by careful naming, or optionally use JSDoc for hinting, but Convex does not *require* TypeScript.

Schema Design and Data Modeling in Convex

Convex takes a **code-first** approach to database schema. You define your data models in code using Convex's schema utilities. When you initialized your project, Convex likely created a stub `convex/schema.ts` (or `.js`) file. In JavaScript, you can write a schema as follows:

```
import { defineSchema, defineTable } from "convex/server";
import { v } from "convex/values";

export default defineSchema({
  users: defineTable({
    name: v.string(),
    email: v.string(),
    createdAt: v.optional(v.number()),
  }),
  messages: defineTable({
    text: v.string(),
    senderId: v.id("users"),    // reference to a user
    timestamp: v.number(),
  }),
  // ... other tables like tasks, contacts, analyticsEvents, etc.
});
```

This example defines two tables: `users` and `messages` with their fields and types (using Convex's built-in validators like `v.string()` and `v.id()` for references). The schema is exported as the default – Convex will use it to enforce that all writes conform to these types [5]. You can add tables for your CRM module (e.g. `contacts`, `deals` tables) or analytics (e.g. an `events` table with event data). Because Convex is a **document-oriented relational** database, you store JSON-like objects (documents) and can relate them via IDs (foreign keys) [6].

Notice the use of `v.id("users")` for the `senderId` field – this enforces that `senderId` must be a valid ID from the `users` table [6]. Such **validation** is a best practice to maintain relational integrity. In fact, Convex encourages adding validators for all function arguments as well (especially for public-facing functions) to ensure you only process expected data types [7] [8]. For example, if you have a mutation to add a new contact, you might declare it like:

```
// convex/contacts.js
import { mutation } from "convex/server";
import { v } from "convex/values";
```

```
export const addContact = mutation({
  args: { name: v.string(), email: v.string() },
  handler: async (ctx, { name, email }) => {
    // ...create a new contact in the database
    return await ctx.db.insert("contacts", { name, email, createdAt:
Date.now() });
  }
});
```

Here the `args` schema ensures `name` and `email` are strings, preventing invalid data from being written [9]. Modeling your data with the correct types up front and using Convex's validators gives you confidence in data consistency – a crucial aspect for scaling reliably.

Using Convex Chef for Rapid Full-Stack Development

Convex's **Chef** is an AI-powered app builder that can scaffold your entire application from a natural language prompt. Think of it as an AI "chef" that, given a recipe (description of the app), cooks up a fully working codebase. You provide a prompt describing the features and data models you need, and Chef generates a complete **React + Tailwind CSS frontend** and a **Convex backend** with database schema, query/mutation functions, auth, file storage, etc., wired up [10] [11]. It's not just code snippets – Chef produces a *running application* with the backend fully integrated. In other words, *"Chef is the only AI app builder that knows backend"*, meaning it will handle those hard parts like database schema and real-time sync that other generators often ignore [12] [11].

Convex Chef interface, where a single prompt (right) can generate a complex app (left preview). In this example, the prompt describes a Notion-like collaborative editor, and Chef sets up the necessary UI and Convex backend.

Using Chef is straightforward: visit the Convex Chef UI and enter your app's description. For instance, you might write *"Build a collaborative tool with a chat room (AI chatbot integration), a real-time analytics dashboard for project metrics, and a simple CRM to track contacts and tasks."* Chef will process this prompt and generate the project files. It will define the tables (e.g. `messages`, `analytics`, `contacts`, etc.) and Convex functions needed [11], as well as the React components for the UI. You can then download the generated code and run it locally. Convex provides a CLI command to **deploy** the backend (e.g. `npx convex deploy`) and you can host the frontend as you normally would (Vercel, Netlify, etc.). In fact, Chef even lets you one-click deploy a preview of the app to Convex's cloud to test it immediately [13].

Schema and Data Modeling with Chef: One of Chef's strengths is that it infers and creates the database schema from your prompt. For example, if your prompt mentions "contacts with names, emails, phone numbers" in a CRM feature, Chef will create a `contacts` table with fields like `name`, `email`, `phone` (using appropriate Convex validators). It similarly sets up any necessary indexes or relations it deduces – e.g. linking "tasks" to "contacts" by storing a contact ID in a tasks table. All this is done using the same `defineSchema` and `defineTable` under the hood, so you can later inspect or modify the schema file as needed. Chef also stubs out Convex **function routing** for you: every query or mutation that the app needs will be defined in the `convex/` directory and exported, and the React code will call them via the Convex client. This means you don't have to manually write API endpoints or hook up WebSocket events – it's all handled by Convex's framework. For instance, Chef might generate a `sendMessage` mutation in `convex/chat.js` and use it in the frontend via

`useMutation(api.chat.sendMessage)` – the *routing* from the UI call to the correct backend function is resolved by Convex's API import, with no extra glue code.

Using Chef Efficiently: To get the most out of Convex Chef, be as clear and specific as possible in your prompt. Describe your data models and features – Chef is surprisingly adept at translating feature requests into schema and code. Once the code is generated, treat it as a starting point: you should **download the project and put it under version control** for further development [14] [15]. From that point, you can modify the Convex functions or add new ones, and use Convex's dev workflow (as described earlier) to deploy updates. Chef's generated code includes helpful comments and even **Cursor** (AI agent) rules to keep your development going smoothly [14]. It's worth noting that Chef is especially powerful for prototyping and initial development – for long-term maintenance you'll continue evolving the code manually (Chef doesn't "round-trip" import your changes) [16]. Still, leveraging Chef can save a huge amount of time setting up the basics of a full-stack app.

Writing Convex Functions and Calling Them (Function Routing)

In Convex, you write **server functions** (queries, mutations, actions) in the `convex/` directory, and they become automatically exposed to your client app through Convex's API. There's no need to write explicit HTTP routes – Convex handles the routing. The function names and file structure act as the identifier. For example, if you have `export const getMetrics = query({...})` in `convex/analytics.js`, the client can call it via `api.analytics.getMetrics()` (or via `anyApi.analytics.getMetrics` in a JS-only setup). Convex's build system will detect this function, ensure it's deployed in the cloud, and the client library will know how to invoke it securely.

Under the hood, Convex assigns each function a unique path based on its module and name. The **client libraries** (for React, Node, plain JS, etc.) include a generated `api` object that has matching functions for each backend export (in TypeScript this is strongly typed; in JS you use the untyped API) [17] [18]. This design eliminates a lot of boilerplate. You don't worry about setting up REST endpoints or GraphQL resolvers – you just call your Convex functions as if they were local. Convex routes the call to the correct function in the cloud, and returns the result (or streams updates, in the case of queries).

Example: Suppose our collaborative app needs a function to log an analytics event when a user completes a task. We can write an **action** (since it's a side-effect, not just a DB write – maybe we also call an external service) in `convex/analytics.js`:

```
import { action } from "convex/server";
import { v } from "convex/values";

export const logEvent = action({
  args: { eventType: v.string(), userId: v.id("users") },
  handler: async (ctx, { eventType, userId }) => {
    // Write to the events table
    await ctx.db.insert("analyticsEvents", { eventType, userId, timestamp:
Date.now() });
    // (Optional) call an external analytics API or webhook
    await fetch("https://third-party-analytics.com/ingest", { method:
"POST", body: JSON.stringify({ eventType, userId }) });
    return "ok";
```

```
  }
});
```

Now any part of our frontend can call this via `api.analytics.logEvent({ eventType: "TASK_COMPLETED", userId })` using the Convex client. The platform will route that call to our `logEvent` action. If the action needs to do more (like trigger other Convex functions), it can. The key takeaway is that **function routing is essentially automatic** – by organizing your Convex functions well (one module per domain or feature, e.g. `chat.js`, `crm.js`, `analytics.js`), you get a clean API that the frontend can use directly.

Convex also supports scheduled function calls (crons) and webhooks, which you define similarly in code (using `cron()` or special naming conventions). These advanced triggers allow you to route events into Convex functions without user action (e.g. a nightly summary email, or responding to an incoming webhook by calling a Convex mutation). All function definitions live in your Convex project, making it easy to trace how data flows through your app.

Real-Time Updates and Reactive Features

One of Convex's superpowers is **realtime reactivity**. Every Convex *query* function you write can be turned into a live query that pushes updates to clients automatically. This means building collaborative, real-time features (like live cursors, analytics dashboards updating in real time, chat messages streaming without refresh, etc.) is much simpler: you just write queries as normal, and the Convex client keeps them up-to-date.

Convex achieves this by maintaining a **WebSocket** connection from each client to the backend. Whenever data in the database changes (due to a mutation or action), Convex's **reactive engine** figures out which query results depend on that data and re-runs those queries, sending the new results to subscribed clients [19] [20]. You don't have to write any special "pub/sub" code or polling logic. For example, if you have a `getTodos` query that fetches all tasks, and one user adds a new task via a mutation, *all other clients* with the `getTodos` subscription will instantly receive the new list of tasks. Convex's backend pushes a diff or new result over the WebSocket, and the Convex client updates the local state (in React, causing a re-render of the component using that query) [20] [21].

How to use this: In React, you'd typically use the Convex React hooks. For example: `const tasks = useQuery(api.todos.getAll);` – this subscribes to the `getAll` query and keeps `tasks` updated. In a plain JS context or Node, you can use `ConvexClient.onUpdate()` as shown earlier. Either way, you simply use the data and the updates flow in.

Best practices for real-time queries: Because any change to underlying data will trigger a re-run of a query, you want to design your queries to be as efficient as possible, especially for high-frequency updates or large data sets. Convex provides **indexes** and supports incremental query filters to optimize this. For instance, if you have an analytics dashboard that shows "active users in the last 5 minutes," you should define an index on the timestamp or user status so that the query only considers recent entries rather than scanning the entire events table on each update. In general, avoid querying huge collections without filters. The Convex docs recommend using `.withIndex` or `.searchIndex` conditions rather than pulling all data and filtering in application code, as the latter can be inefficient for large sets [22] [23]. In fact, a rule of thumb is to **only use** `.collect()` **(which gathers the full result set) for small numbers of documents**. If there could be, say, thousands of records, use pagination or specific queries. As the docs note: if a query returns an unbounded number of documents, it not only transfers a lot of data (affecting bandwidth costs) but also means *any change to*

any of those documents triggers a re-run [24]. By narrowing queries with indexes or limits, you reduce the reactive workload and improve performance.

For example, instead of a query that subscribes to *all* analytics events (which could be enormous), you might have the client specify a time range or use a Convex **search query** for just today's events. Or use `.paginate()` for infinite scroll tables. Convex's reactivity is powerful, but you as the developer can help it out by structuring data and queries intelligently. In practice, common real-time features like collaborative text editing or live dashboards map well to Convex's model: each document or record can be observed and updated individually. For instance, a collaborative document in Notion-style app could be a record in a `documents` table; users editing it would update its content via mutations, and anyone viewing it would be subscribed via a query – voila, real-time sync of document content (Convex will send patches with changed fields).

Finally, remember that **mutations are transactional** and **atomic** in Convex [25] [26]. This means you can safely perform multi-step updates in a single mutation (e.g. update two related tables) and Convex will ensure consistency (retrying on conflicts if needed). This is great for real-time apps because you never see partial updates – either all clients see the full committed state or nothing if a conflict occurs and is retried. Combined with the real-time push, this yields a very robust collaborative experience (no half-saved states or out-of-sync clients).

Integrating an AI Chatbot with Convex

An **AI-powered chatbot** can greatly enhance a collaborative tool, providing users with assistance, insights, or automation. In our architecture, we'll integrate a chatbot by using Convex **actions** (serverless functions that can call external APIs) to interface with an AI service (like OpenAI's GPT-4). We can also take advantage of Convex's new **AI Agents** framework for a higher-level abstraction.

Basic approach (Actions): You can create an action that sends the user's query to an AI API and stores the response. For example, define a function `askAI` in `convex/chatbot.js`:

```
import { action } from "convex/server";
import { v } from "convex/values";
// Assume OPENAI_API_KEY is set in Convex environment variables
import fetch from "node-fetch";  // node-fetch or similar to call external
API

export const askAI = action({
  args: { prompt: v.string() },
  handler: async (ctx, { prompt }) => {
    // Call the OpenAI API (or other AI service)
    const response = await fetch("https://api.openai.com/v1/chat/
completions", {
      method: "POST",
      headers: { "Authorization": `Bearer ${process.env.OPENAI_API_KEY}` },
      body: JSON.stringify({ model: "gpt-4", messages: [{ role: "user",
content: prompt }] })
    });
    const data = await response.json();
    const answer = data.choices[0].message.content;
```

```
    // Save the Q&A in a messages table for history
    const message = { role: "assistant", text: answer, createdAt:
Date.now() };
    await ctx.db.insert("messages", message);
    return answer;
  }
});
```

In the frontend, you'd call `api.chatbot.askAI({ prompt: userQuestion })` and await the result. The action will invoke the external API and return the answer. By storing the message in a Convex table (`messages`), you enable real-time updates: other users (or the same user on another device) can subscribe to the `messages` query and see the bot's answer appear live. This setup essentially gives you a persistent chat history (the CRM module could even tie these answers to specific contacts or tickets, if that's relevant).

Advanced approach (Convex AI Agents): Convex provides an **Agent** component that abstracts the pattern of long-running conversations with AI models, including tools and memory [27] [28]. Using Agents, you can set up a chatbot with built-in context, vector search for knowledge base articles, and more – all within Convex. For example, you might create an agent in an action:

```
import { Agent } from "@convex-dev/agents";
import { openai } from "@ai-sdk/openai";  // Convex's AI SDK
import { components } from "./_generated/api";
import { action } from "convex/server";

const supportAgent = new Agent(components.agent, {
  name: "SupportBot",
  chat: openai.chat("gpt-4"),  // model to use
  instructions: "You are a helpful assistant for our collaborative tool.",
  tools: { /* define any tools the agent can use, e.g., database lookup */ }
});

export const startChat = action({
  args: { prompt: v.string() },
  handler: async (ctx, { prompt }) => {
    const { thread, threadId } = await supportAgent.createThread(ctx);
    const result = await thread.generateText({ prompt });
    return { threadId, reply: result.text };
  }
});
```

This uses Convex's built-in agent integration [29] [30]. When `startChat` is called, it starts a new conversation thread with the AI, gets a response, and returns it (optionally storing the thread in Convex for continued dialog). The advantage of this method is that Convex manages the conversation state, and you can later call `supportAgent.continueThread(threadId)` to continue the same conversation, with full history and even **context from your database** (Convex Agents automatically include previous messages and can do retrieval-augmented generation via vector search in your Convex data [31] [32]). Essentially, Convex Agents turn your backend into a sophisticated chatbot orchestration engine, all in JS/TS code.

For our collaborative app, a simple implementation might suffice (using actions to call OpenAI), but it's good to know that as your needs grow (say, integrating the chatbot with the CRM data to answer questions about client info, or giving it tools to create analytics reports), Convex's AI primitives can scale up to that complexity.

Implementing Real-Time Analytics Dashboards

The tool we're building should include real-time analytics – for example, a dashboard showing usage statistics, project progress, or other metrics that update live as users interact. With Convex, real-time analytics become easier because any metric stored or computable in the database can be **queried reactively**.

Capturing events: First, decide what events or data points you need to track. You might have a Convex mutation like `recordEvent` that inserts a document into an `analyticsEvents` table (as shown earlier with `logEvent`). This could capture things like `{ eventType: "TASK_COMPLETED", user: <ID>, timestamp: <time> }` or any domain-specific event (e.g. "new note created", "comment added"). Each time such a mutation is called, a new event is stored.

Computing analytics: For real-time display, you can write Convex *query functions* that aggregate these events. For instance, a query to get the count of tasks completed today per user might look like:

```
// convex/analytics.js
import { query } from "convex/server";
import { v } from "convex/values";

export const tasksCompletedToday = query({
  args: { },
  handler: async (ctx, args) => {
    const sinceMidnight = new Date().setHours(0,0,0,0);
    return await ctx.db.query("analyticsEvents")
                .withIndex("by_eventType_and_time", q =>
                    q.eq("eventType", "TASK_COMPLETED").gte("timestamp",
sinceMidnight))
                .collect();
  }
});
```

Suppose we created an index on `analyticsEvents` for `(eventType, timestamp)` – this query will efficiently fetch all "TASK_COMPLETED" events from today. On the client, use something like `const events = useQuery(api.analytics.tasksCompletedToday);`. Now the UI will receive the up-to-date list of today's completed-task events, and you could derive a count or display a chart. As new events come in (users complete tasks), the query results will refresh in real time.

For heavier aggregations (like computing a sum or grouping by user), you have a couple of options: - **Do it in the query**: Convex allows you to do JS computations in the query function after fetching data. If the result set is reasonably bounded (e.g. number of events today isn't huge), you can `.collect()` them and then use JS to count or group. If you expect the data to grow large, it's better to maintain a **denormalized counter**. For example, each time a task is completed, also increment a counter in a `userStats` table for that user. That way a query can just pull the current counts from `userStats`

which is a single document per user (much lighter). - **Use Convex db functions**: There's also an `db.count()` and other helpers if needed, or you can implement counters via Convex's atomic update (using `ctx.db.patch` or similar to increment fields).

The key is that once you have the metric available in a Convex query, feeding it to a real-time dashboard is trivial – just subscribe and render. Want a chart of active users? Maintain an "active" status (maybe via a heartbeat or connection event) and query all active user records. Want real-time sales totals? Have a table that sums sales and update it on each order, or run a query with `.sum()` if your data adapter supports it.

Reactive charts and filters: Convex queries can take parameters, so you can even have interactive analytics. For instance, a query `salesByRegion(regionId)` that returns sales for a given region, and your UI can subscribe to `api.analytics.salesByRegion({ regionId: X })`. When the user switches region, it'll subscribe to a different query. Convex will manage unsubscribing from the old one and subscribing to the new one behind the scenes. Each query's reactivity is scoped to its inputs.

One thing to watch in analytics: as mentioned, avoid **unbounded queries** that return huge sets of data. If you need historical analysis over millions of events, consider using Convex's **CSV export** or connecting a dedicated warehouse. But for real-time metrics in an app (usually focused on recent or summary data), Convex can handle it fine, and the auto-update mechanism means your dashboard is always fresh without manual polling.

Building a CRM Module with Convex

The CRM aspect of our app involves managing contacts, companies, interactions, etc., in a collaborative way. Let's outline how you could implement a simple CRM module on Convex:

- **Data modeling:** Suppose we want Contacts, Companies, and Deals. We'd create tables in our Convex schema for each:
- `contacts` with fields like `{ name: v.string(), email: v.string(), company: v.id("companies"), ... }`
- `companies` with fields `{ name: v.string(), industry: v.string(), ... }`
- `deals` with fields `{ title: v.string(), value: v.number(), contact: v.id("contacts"), stage: v.string(), ... }`

This sets up relationships: each contact links to a company by ID, each deal links to a contact (or maybe to a company or both). Convex being document-based means you don't need to join tables via foreign keys in queries; you often fetch related data by performing multiple queries or using stored references. For example, you might have a query `getDealsForCompany(companyId)` that first fetches all contacts of that company, then fetches all deals for those contacts. Or you could denormalize by storing `companyId` on deals directly for quicker lookup.

- **Functions:** Write Convex mutations for CRUD operations:
- `createContact`, `updateContact` – to add/edit contact info
- `createDeal`, `moveDealStage`, etc. – to manage deals
- Possibly `linkContactToCompany`, though if you include company in contact data you might just update the contact.

Each mutation should validate its inputs (e.g. ensure emails are string, perhaps add regex validation, ensure IDs exist by maybe querying the referenced table). The transactional nature of Convex

mutations allows you to do things like, when a new contact is created, increment a `numContacts` field on the corresponding company document in the same mutation.

- **Realtime collaboration:** Because all data is in Convex, if two salespeople are looking at the contacts list and one adds a contact, the other's UI (subscribed via a `listContacts` query) will update immediately. Similarly, if a deal's stage is changed (via a mutation), anyone querying the deals for that company will see it live. Convex ensures consistency – for instance, if a contact is deleted, you might have a trigger (or just handle it in the mutation) to also remove or nullify related deals. Clients will then get updates that those deals changed or were removed.

- **Access control:** In a real CRM, you'd consider permissions (Convex has an auth system you can integrate with OAuth or custom tokens). You can use the `ctx.auth` object in Convex functions to check the user identity and enforce that, say, only team members of a project can see its deals. While an in-depth auth setup is beyond scope here, Convex does support role-based rules in your functions or using a **public vs private function** distinction (by default all functions are private to authenticated users). You'd implement checks in your mutation handlers (throwing an error or doing nothing if the user isn't allowed). Because everything goes through your code (no direct table access from client), it's straightforward to add these checks.

In summary, building a CRM module on Convex is mostly about defining your data schema and writing a set of mutations/queries for the operations. The front-end can then use these via Convex hooks (for listing, details, etc.) and the experience will be real-time for all users. For example, you could have a contact detail view that uses `useQuery(api.contacts.getContact, { id })` to fetch a contact's info (with live updates if someone edits that contact elsewhere), and maybe another query `useQuery(api.deals.dealsByContact, { contactId: id })` to get related deals. Any new deal added for that contact (via `api.deals.createDeal`) would trigger the dealsByContact query to refresh [33] [20], updating the UI instantly.

Scalability: How Convex Scales from Small Teams to Enterprise

A critical consideration for any collaborative app is whether it can scale as your user base grows. Convex's architecture is built with scalability in mind, so that a small prototype can later support enterprise-level usage without a complete rewrite. Here's how Convex handles scaling:

- **Serverless auto-scaling:** Convex operates a multi-tenant cloud service that automatically scales the underlying resources based on load. As a developer, you **don't need to manage servers, database instances, or load balancers** – Convex takes care of that [34]. You focus on writing your functions and data logic. Convex's team (comprised of ex-Dropbox engineers) designed the system to handle very large scale (they've dealt with exabyte-scale systems before) [35]. In practice, this means your app can go from 10 daily users to 100k daily users, and while your usage costs will rise (addressed in the next section), you won't hit an architectural wall where things break down. You won't suddenly need to shard your database or rewrite your real-time sync – Convex abstracts those scaling challenges.

- **Horizontal function execution:** Initially, each Convex deployment ran on a single backend process, which could handle a certain number of concurrent function executions. Convex recently introduced **horizontal scaling of function runners** via a service codenamed *Funrun* [36]. Essentially, your Convex backend can now call out to a pool of worker processes to run your functions in parallel, beyond the limits of a single CPU. This change removed prior concurrency limits (like V8's thread cap) [37] [38]. For your app, this means even if hundreds of users trigger

functions simultaneously (e.g. a burst of chat messages or analytics events), Convex can distribute the load across multiple machines. The data consistency is still maintained (all writes funnel through the transaction protocol) but the compute scales out.

- **Database scaling:** Convex's cloud uses a SQL (MySQL) underlay for persistence [39], and the team manages the scaling of this as well. For most use cases, the provided database will handle a lot of data (gigabytes to tens of GBs easily, as seen in the included quotas). If you approach those limits, it likely means you have an extremely successful app! In enterprise scenarios, Convex can work on a more tailored solution – they've indicated an enterprise plan that could involve larger isolated clusters or additional performance guarantees [40] [41]. There's also the option to **self-host** Convex's open-source backend on your own infrastructure (e.g. on a beefy Postgres cluster) if needed [42], although then you assume responsibility for scaling it.

- **From dev to prod:** For a small team or startup, Convex's free tier supports up to 1 million function calls per month, which is plenty for early-stage. As you grow, you'd move to the **Professional** tier which includes higher fixed quotas and pay-as-you-go scaling beyond them. Importantly, the *architecture* of your app doesn't change – you might tweak function implementations to be more efficient, add indexes as data grows (for query speed), or use caching where appropriate, but you won't need to redesign how clients connect or how state syncs. Convex's real-time engine and function routing remain the same at 100 users or 100k users. This allows you to **iterate quickly** at the start (focus on features, not infra) and trust that Convex will scale out behind the scenes when needed.

To give a concrete sense of scale: imagine our app in enterprise use – thousands of concurrent users editing documents, chatting, logging CRM updates. Convex would have many open WebSocket connections (one per active client, which it can handle), and lots of function calls per second. The Convex service would simply spawn more function runner instances to handle the load, and ensure the database has read replicas or other optimizations as needed. All this is opaque to us, except maybe we'd notice higher throughput and of course, increased usage metrics in our Convex dashboard. But the experience for users would remain real-time and consistent. Convex's philosophy is that developers *"don't have to worry about scale or rearchitect the system when the database starts falling over"* [43] – the platform itself is built to avoid that scenario by design.

Understanding Convex's Pricing Model

Convex uses a **usage-based pricing model**. This means you pay based on the resources your app consumes: primarily the number of function calls, the amount of data stored, and the bandwidth (data egress) used. This model can be very cost-efficient for small apps (you pay nothing or very little for low usage) and scales with your success as more people use the app. Let's break down the key components driving Convex costs:

- **Function Calls:** Every invocation of a query, mutation, or action counts as a function call. Convex includes a generous allowance in the free tier (e.g. 1 million calls per month on the Starter plan) [44]. Beyond that, you pay a small amount per million calls (on the Starter tier it's about $2.20 per million calls; on Professional it's $2.00 per million, slightly cheaper) [45] [46]. For perspective, a million calls is a lot – even if each user action triggered 10 calls, that's 100k user actions. Convex counts all function executions, whether they are queries (reactive fetches) or mutations/actions (writes or external calls). Reducing unnecessary calls (e.g. debouncing frequent actions, combining some logic into single calls) can optimize this cost, but generally function calls are cheap.

- **Data storage:** You are billed for the total data your app stores in the Convex database (and any files or vector embeddings, if you use those features). The free tier gives 0.5 GB of DB storage, 1 GB file storage, etc. [44] . Extra database storage is around \$0.20 per GB per month [47] – in line with cloud DB storage costs. File storage is even cheaper (\$0.03/GB/mo) [48] . For our app, unless you're storing a lot of large text or files, the database size will mainly grow with number of records. Text data (notes, messages) is usually not that heavy; file attachments (if you allow uploads in the app) might need monitoring. Convex's vector storage (for AI embeddings) is \$0.50/GB/mo [49] , relevant only if you store embedding indices for AI features.

- **Bandwidth (egress):** This is often what "data egress" refers to – the amount of data leaving Convex servers to your clients. Convex measures database bandwidth (data read out of the DB by queries) and file bandwidth (files downloaded). Each has its pricing (around \$0.20 per GB for DB reads [50] and \$0.30 per GB for file downloads [51]). What does this mean practically? If your queries send a lot of data to users (e.g. a query returning 1000 records of 1KB each = ~1 MB every time it runs), and this happens frequently, that will count against the bandwidth quota. This is another reason to use indexes and limit query results to just what you need – it not only improves performance but also saves bandwidth (and money). For instance, sending incremental updates (like "just the new message" rather than the entire chat history each time) keeps bandwidth low. Convex's real-time diffing usually ensures that after the initial query load, only changes are sent, which mitigates bandwidth usage for reactive queries. Still, if you have very data-heavy use (like transferring many images or large documents), keep an eye on file bandwidth – 1 GB covers a lot of text, but a few hundred high-res images could surpass that.

- **Computation (Action compute time):** Convex measures the CPU time for **Node.js actions** (the ones that can run longer and do external calls). They give some free GB-hours per month (20 GB-hours on Starter, 250 on Professional) [44] [52] . "GB-hour" here means 1 GB of memory for 1 hour (or 2 GB for 0.5h, etc.). Most actions run in a few milliseconds to a few seconds, so this usually isn't a big factor unless you do heavy data processing in actions. It's priced around \$0.30 per GB-hour over the free quota [53] . For example, if your AI chatbot action holds 512MB memory and takes 2 seconds per call, that's 1/1800 of an hour * 0.5 GB ≈ 0.00028 GB-hours per call – you could do many of those before hitting even 1 GB-hour.

- **Chef Tokens:** Convex Chef usage is also metered – the free tier includes some number of "Chef tokens" (85k for Starter) [54] . These tokens are essentially the prompt/response tokens fed to the LLM when using the Chef AI generator. If you heavily use the AI code gen, you might run out of free tokens and then it's about \$11 per million tokens [55] . However, this cost is usually negligible unless you are programmatically using Chef or generating many apps. For building our app, you'd likely use Chef interactively a few times (well within the free allotment).

In summary, **Convex's pricing is usage-driven**: if your app is lightly used, you pay little or nothing; if it's heavily used, the costs grow in proportion to the value delivered (more users/collaboration). For planning purposes, Convex provides a cost calculator to estimate your bill based on expected calls, storage, etc [56] . As a developer, you should design with efficiency in mind (to minimize unnecessary function calls and data transfer), but you don't need to obsess over minor costs – the pricing is quite reasonable (e.g., \$2 per million calls, \$0.20/GB). One nice aspect is that **scaling is linear and granular**: you don't jump from free to a sudden \$1000/month server cost – you pay maybe a few dollars as you exceed free limits, then tens, and so on, scaling with actual usage.

Comparing Convex's Pricing to Notion's Pricing Model

It's useful to contrast Convex's usage-based pricing with the pricing model of a platform like **Notion**, since Notion is a well-known collaborative tool. Notion (as a service) uses a **per-user subscription** pricing model. For example, Notion's *Plus* plan is around \$10 per user per month (billed annually, about \$12 if monthly), and the *Business* plan is about \$18 per user per month [57]. These fixed fees grant each user essentially unlimited use of the app's features – there's no metering of how many edits or how much data one user uses (within generous limits). The cost drivers for Notion are thus **number of users and plan tier**, rather than raw usage. A team of 10 on Plus would pay about \$120/month no matter if they use the workspace heavily or sparingly.

Notion's pricing bundles the infrastructure costs into that user fee. For instance, paid plans allow larger file uploads (up to 5 GB file uploads on Plus, vs. 5 MB on the free plan) and unlimited blocks of content [58]. The rationale is that a paying user is expected to perhaps consume more storage and server load, so the higher plan covers that. However, as an end-user or org, you don't directly see a charge for "we made 100,000 edits this month" – you just pay the flat per-seat price. In contrast, an app built on Convex would not have a built-in per-seat cost (you could have 100 users idle and pay nothing), but if those users perform a lot of actions (function calls) or store a lot of data, the usage costs would increase accordingly.

What drives Notion's costs? Internally, even though Notion charges per seat, their operating costs likely scale with usage in a similar way: hosting data, syncing changes in real-time (lots of operations per second), and now AI features (which call expensive language models). They've chosen to average this out and charge a flat rate to customers. Notion's AI addon, for example, is a fixed \$8/person on Plus, \$15 on Business [59], which gives "unlimited" AI queries to the user – in reality they'll have some fair use policy or token limits behind the scenes [60], but the user isn't billed per prompt. This is a key difference: **Convex passes usage costs directly to the developer, whereas Notion hides it behind a subscription.**

For a developer/business deciding between building your own app on Convex vs using a tool like Notion, the cost structure differs: - With Convex, you pay for what your users actually do (which can be very efficient if, say, you have many occasional users). If your app becomes as complex as Notion and extremely popular, you'd be paying Convex for the high usage – but presumably you'd have a business model (perhaps also subscriptions) to cover that. Convex's pricing is quite transparent and in many cases cheaper at scale than per-seat pricing. For example, 100 users on Notion Business is ~\$1500/month, whereas 100 active users on a Convex app might easily cost much less if each user isn't pushing massive data constantly. - Notion's pricing, being fixed, is predictable and simpler on the surface. The cost is justified by the value of the app, not the literal compute/storage use. They also provide enterprise features (SAML SSO, advanced permissions) as part of higher tiers [61]. If you build your own app on Convex, you'd have to implement those features, but you gain flexibility and potentially cost savings if your usage pattern is lower.

In essence, Convex is like paying for cloud infrastructure (serverless backend) on a pay-as-you-go basis, while Notion is a finished SaaS product where you pay for user licenses. If your collaborative tool aims to be a Notion-like service offered to others, you might adopt a hybrid approach: use Convex's usage-based costs as your backend expenses and perhaps charge your customers per seat as well. In that scenario, it's important to note which actions drive your Convex costs (function calls and egress). For instance, a very collaborative workspace with constant edits will generate many function calls (Convex costs go up, while Notion's costs to the customer stay flat). But also, Convex's pricing has ceilings in higher tiers or enterprise deals – you might negotiate or self-host if you reach truly massive scale.

To conclude this comparison: **Notion's cost to users is fixed per user, whereas Convex's cost to developers is variable by usage.** Each approach has its merits. For our guide's purposes, building on Convex means we need to monitor usage to manage cost, but we also get fine-grained control to optimize (e.g. caching certain data to reduce calls, pruning old data to save storage, etc.). And if our app is small, we might run on Convex's free tier or just a few dollars a month – which is something not possible with Notion (which would charge at least \$10/month for one user on a paid plan).

Conclusion and Best Practices

By using Convex with a JavaScript-only approach, we can rapidly develop a full-stack collaborative tool with minimal overhead. We harnessed Convex's **Chef** to jump-start the project, automatically generating schema, functions, and a reactive UI from a prompt. We designed our data models directly in code, leveraging Convex's document-relational schema to link data elegantly. With Convex's built-in **real-time sync**, features like the AI chatbot, live analytics, and CRM updates all propagate instantly to every user – boosting collaboration without complex client-side state management.

Best practices summary:

- *Keep functions focused and validated:* Write small, purposeful Convex functions (queries for reads, mutations for writes, actions for external calls) and use `v.*` validators for all inputs [9]. This ensures data integrity and security.
- *Optimize for reactivity:* Use indexes for queries that filter large tables and paginate or limit results for heavy data lists [24]. This reduces bandwidth and keeps updates snappy. Remember that any change to any data a query returns will trigger a re-run – so avoid queries that return more data than needed.
- *Embrace Convex's real-time model:* Structure your app around Convex subscriptions (via `useQuery` or `onUpdate` callbacks). Avoid designing manual polling. Let Convex push data – it's more efficient and provides a better UX. For example, instead of an expensive "refresh" button that reloads all data, rely on the reactive queries to update the UI automatically when something changes.
- *Leverage AI where it adds value:* Our chatbot integration showed how to use Convex actions/agents to integrate AI. Keep those calls server-side (so you don't expose API keys) and store conversation context in Convex if you need memory. Convex's agent system can manage long conversations and tool usage – consider it when your AI needs grow beyond simple Q&A.
- *Monitor usage and scale gradually:* Use Convex's dashboard to observe function call counts and performance. This will highlight any "hot" functions that might need optimization (e.g., a query being called too often or returning too much data). With scaling handled by Convex, your job is mainly to ensure you're not doing wasteful work. If you hit usage limits, you can upgrade the plan or refactor the code causing high usage. Write idempotent, conflict-resistant mutations so that even under high concurrency, your data remains consistent (Convex handles retries of transactions automatically [62]).
- *Cost awareness:* While you don't want premature micro-optimizations, it's good to be aware of what operations could drive cost. For instance, sending a large file to every client repeatedly would hit bandwidth costs – instead, maybe send a reference and have clients fetch it only once or use caching. In our app, typical operations like adding a contact or logging an event are lightweight. The biggest costs might come from heavy analytics queries or extremely active chatrooms, but those are also signs of a healthy app. Plan your pricing (if you charge users) to align with your Convex costs (e.g., if you anticipate each user triggers \$0.50 of Convex costs monthly, you ensure your subscription fee or revenue per user is higher).

In building this AI-native collaborative tool, we saw that **Convex simplifies the full-stack development**: the backend is written in the same spirit as frontend code (just JavaScript functions), and state synchronization is handled for us. Our architecture remains clean and minimal – essentially a client app + Convex backend – yet it's powerful enough to handle real-time editing, AI interactions, and multi-user data management at scale. By following the practices outlined and understanding how Convex scales and charges for usage, we can confidently grow our application from a prototype for a small team into a robust platform for an entire enterprise. The result is a highly responsive, collaborative tool without the bloat of heavy infrastructure, truly *AI-native* and ready for the demands of modern users.

[63] [43]

---

[1] [2] [5] [9] [19] [20] [21] [33] My Journey with Convex. From API Routes to Real-Time Magic: My... | by Venkatesh Sundarasan | Aug, 2025 | Medium
https://medium.com/@venkateshsundarasan/my-journey-with-convex-4274855f9133

[3] [4] [17] [18] JavaScript | Convex Developer Hub
https://docs.convex.dev/client/javascript

[6] [26] [39] [63] Convex Overview | Convex Developer Hub
https://docs.convex.dev/understanding/

[7] [8] [22] [23] [24] Best Practices | Convex Developer Hub
https://docs.convex.dev/understanding/best-practices/

[10] [11] [12] Convex Chef: Unlocking AI Potential with Innovative Tools
https://skywork.ai/skypage/ko/Convex-Chef:-Unlocking-AI-Potential-with-Innovative-Tools/1972570729255464960

[13] [14] [15] [16] Chef | Convex Developer Hub
https://docs.convex.dev/chef

[25] [34] [35] [36] [37] [38] [43] [62] How We Horizontally Scaled Function Execution
https://stack.convex.dev/horizontally-scaling-functions

[27] [28] [29] [30] [31] [32] AI Agents | Convex Developer Hub
https://docs.convex.dev/agents

[40] [41] [42] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [56] Convex Pricing, Features & Best Alternatives (2025)
https://www.srvrlss.io/provider/convex/

[54] [55] Plans and Pricing
https://www.convex.dev/pricing

[57] [58] [61] Notion Pricing - Plans
https://www.joinsecret.com/notion/pricing

[59] [60] AI Agent Pricing 2025: Complete Cost Breakdown ($8-20/User) - Blog
https://www.mathewtamin.com/blog/notion-ai-pricing-2025-complete-cost-breakdown-7-better-alternatives