EcomLabs

STEP-BY-STEP GUIDE

Function calling explained with a shopping cart

The full example in Node and TypeScript: three cart tools with strict: true, the tool_use and tool_result loop, errors with is_error, a test that runs without an API key, and the security checklist.

FOR	Stores and teams that want an AI assistant to find products, fill the cart and quote shipping
TESTED WITH	Node 22.22.3 · @anthropic-ai/sdk 1.6.0 and tsc --strict · mocked client returning documented-shape responses, no API calls
UPDATED	November 19, 2026
DOCS	platform.claude.com/docs/en/agents-and-tools/tool-use/overview
ARTICLE	ecomlabs.dev/g/tools-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you will build
Step 01	The full file
Step 02	Test it and use it
When you finish	Security checklist

BEFORE YOU START

What you will build

A shopping assistant with function calling: the model finds products, adds them to the cart and quotes shipping by calling three functions in your code. It all fits in `store.ts`. We tested it on Node 22.22.3, which runs `.ts` files directly, with `@anthropic-ai/sdk` 1.6.0. In a new folder: `npm init -y`, `npm pkg set type=module` and `npm install @anthropic-ai/sdk`. The API shapes follow Anthropic's tool use documentation (November 19, 2026); the catalog, prices and shipping rule are examples.

STEP 01

The full file

Save this as `store.ts`: the sample catalog, the three tools with `strict: true`, the function that runs and validates them, and the loop.

```
// store.ts: function calling with three cart tools (Node 22 + @anthropic-ai/sdk)
import Anthropic from "@anthropic-ai/sdk";

// Sample catalog, prices in cents (USD). In your store it comes from the database.
const CATALOG = [
  { sku: "MUG-12OZ", name: "Ceramic mug 12 oz", price: 1400, stock: 12 },
  { sku: "COFFEE-1LB", name: "Whole bean coffee 1 lb", price: 1800, stock: 0 },
  { sku: "PRESS-34OZ", name: "French press 34 oz", price: 3900, stock: 3 },
];
const MAX_PER_ITEM = 10;
const MAX_TURNS = 6;
export type Cart = Map<string, number>; // sku -> quantity, one per session

// Strict schema: every field required, no extra fields.
const tool = (name: string, description: string, properties: Record<string, object>): Anthropic.Tool => ({
  name,
  description,
  strict: true,
  input_schema: { type: "object", properties, required: Object.keys(properties), additionalProperties: false
},
});

export const tools: Anthropic.Tool[] = [
  tool(
    "search_products",
    "Searches the catalog by product name. Returns up to 5 matches with sku, name, price in cents (USD) and
units available. Use it before adding anything to the cart, to get the exact sku. It cannot look up orders or
the cart.",
    { query: { type: "string", description: "Text to search for, for example: mug" } },
  ),
  tool(
    "add_to_cart",
```

```
    "Adds a product to the current session's cart. Needs the exact sku returned by search_products and a whole quantity from 1 to 10. If there is not enough stock, it returns an error with the units available. The store sets the price; it is not an input.",
    {
      sku: { type: "string", description: "Exact SKU, for example MUG-12OZ" },
      quantity: { type: "integer", description: "Units to add, from 1 to 10" },
    },
  ),
  tool(
    "calculate_shipping",
    "Calculates shipping for the current cart to a US ZIP code. Use it only when the cart has items. Returns subtotal, shipping and total in cents. It does not place the order or charge anything.",
    { zip_code: { type: "string", description: "5-digit ZIP code" } },
  ),
];

// An error the model can fix: it goes back as a tool_result with is_error.
class ToolError extends Error {}

const subtotal = (cart: Cart) =>
  [...cart].reduce((s, [sku, n]) => s + CATALOG.find((p) => p.sku === sku)!.price * n, 0);

function runTool(name: string, input: unknown, cart: Cart): string {
  if (typeof input !== "object" || input === null) throw new ToolError("Input must be an object.");
  const args = input as Record<string, unknown>;
  switch (name) {
    case "search_products": {
      const q = String(args.query ?? "").trim().toLowerCase();
      if (!q) throw new ToolError("The query is empty.");
      const found = CATALOG.filter((p) => p.name.toLowerCase().includes(q)).slice(0, 5);
      return JSON.stringify(found.map(({ sku, name, price, stock }) => ({ sku, name, price, available: stock })));
    }
    case "add_to_cart": {
      const product = CATALOG.find((p) => p.sku === args.sku);
      if (!product) throw new ToolError(`SKU ${String(args.sku)} does not exist. Use search_products to get it.`);
      const quantity = args.quantity;
      if (typeof quantity !== "number" || !Number.isInteger(quantity) || quantity < 1 || quantity > MAX_PER_ITEM) {
        throw new ToolError(`Quantity must be a whole number from 1 to ${MAX_PER_ITEM}.`);
      }
      const total = (cart.get(product.sku) ?? 0) + quantity;
      if (total > product.stock) throw new ToolError(`Only ${product.stock} units of ${product.sku} are available.`);
      cart.set(product.sku, total);
      return JSON.stringify({ sku: product.sku, quantity: total, subtotal: subtotal(cart) });
    }
    case "calculate_shipping": {
      const zip = String(args.zip_code ?? "");
      if (!/^\d{5}$/.test(zip)) throw new ToolError("The ZIP code must have 5 digits.");
```

```
      if (cart.size === 0) throw new ToolError("The cart is empty.");
      const sub = subtotal(cart);
      const shipping = sub >= 7500 ? 0 : 799; // illustrative rule: free shipping from $75
      return JSON.stringify({ zip_code: zip, subtotal: sub, shipping, total: sub + shipping });
    }
    default:
      throw new ToolError(`There is no tool called ${name}.`);
  }
}

const SYSTEM =
  "You are the shopping assistant of an online store. Use the tools to find products, add them to the cart and
calculate shipping. Never make up prices or SKUs. If a tool returns an error, fix the call or tell the
customer what happened.";

export async function converse(client: Anthropic, question: string, cart: Cart) {
  const messages: Anthropic.MessageParam[] = [{ role: "user", content: question }];

  for (let turn = 0; turn < MAX_TURNS; turn++) {
    const res = await client.messages.create({
      model: "claude-opus-5-5",
      max_tokens: 16000,
      system: SYSTEM,
      tools,
      messages,
    });

    if (res.stop_reason !== "tool_use") {
      // end_turn is the normal answer; refusal or max_tokens go back to the caller.
      const text = res.content.flatMap((b) => (b.type === "text" ? [b.text] : [])).join("\n");
      return { text, stop: res.stop_reason, messages };
    }

    messages.push({ role: "assistant", content: res.content }); // unchanged, thinking blocks included
    const results: Anthropic.ToolResultBlockParam[] = [];
    for (const block of res.content) {
      if (block.type !== "tool_use") continue;
      try {
        const output = runTool(block.name, block.input, cart);
        results.push({ type: "tool_result", tool_use_id: block.id, content: output });
      } catch (err) {
        if (!(err instanceof ToolError)) console.error(block.name, err); // log the real cause, don't send it
        const message = err instanceof ToolError ? err.message : "Internal store error. Try again later.";
        results.push({ type: "tool_result", tool_use_id: block.id, content: message, is_error: true });
      }
    }
    messages.push({ role: "user", content: results }); // every result, in one message
  }
  throw new Error(`No final answer after ${MAX_TURNS} turns.`);
}
```

- `ToolError` separates what the model can fix (it goes back with its message) from your own failures (a generic message goes back and the detail goes to your logs).
- The cart is a parameter of `converse`, not of the tools: your server sets it from the session.
- `MAX_PER_ITEM` lives in the code because strict mode does not support `minimum` or `maximum` (the API answers 400). Prices are integers in cents, so totals pick up no floating-point noise.

STEP 02

Test it and use it

Without spending. The client is a parameter, so a `node --test` test can pass in a fake one: `{ messages: { create: async () => replies.shift()! } } as unknown as Anthropic`, where `replies` are messages in the documented shape (`stop_reason: "tool_use"` and `tool_use` blocks with `id`, `name` and `input`). Test the happy path (two mugs, total 2800 + 799 cents) and the errors: quantity 50, an invented SKU, a four-digit ZIP code and a model that never stops asking for tools.

With the API. `await converse(new Anthropic(), "Add two mugs and quote shipping to 10001", cart)`, with the key in `ANTHROPIC_API_KEY` and one cart per session. Check `stop` before you show `text`: on `refusal` or `max_tokens`, you decide what the customer sees. Do not force the tool with `tool_choice` set to `any` or `tool`: on Claude Opus 5.5 it returns a 400.

> **CAUTION**
> A test with a fake client checks your code, not the model. Before opening it to customers, run real conversations in a test environment and review in your logs which tools it called and with which arguments.

WHEN YOU FINISH

Security checklist

- [ ] Price, discounts, cart and customer come from your server; no tool receives them from the model.
- [ ] Every `input` is treated as user input: the SKU exists, the quantity is in range, there is stock (and overselling is solved at payment time).
- [ ] No tool charges, refunds or confirms orders; the customer pays at checkout.
- [ ] Fixable errors go back with `is_error: true`; internal ones with a generic message, no stack traces or keys.
- [ ] Third-party content (reviews, supplier descriptions) goes only inside `tool_result` and is treated as untrusted.
- [ ] The loop has a turn cap and only runs tools when `stop_reason` is `tool_use`.
- [ ] There is a per-session rate limit, you log every call, and the API key lives only on the server.