ChatGPT

Project Idea: “CompliBot” – An Agent for Protocol Compliance Validation

We propose **CompliBot**, a solo-built E2B-enabled agent (in TypeScript/Next.js) that checks web protocols (especially HTTP and related specs) for correct usage against official standards. In the demo, the user enters a target (e.g. a URL or API endpoint), and CompliBot runs inside an E2B sandbox, invoking real tools via Docker’s MCP catalog, to inspect responses and verify compliance with RFCs. For example, it will fetch HTTP responses and ensure security headers (like HSTS, CSP, CORS) are present and correctly formatted; it can also optionally introspect a GraphQL schema or simulate a WebSocket handshake and check it against the spec. This idea emphasizes technical depth (e.g. parsing headers and protocol frames rigorously) rather than a common generic agent. It leverages E2B for safe code execution and Docker MCPs for real-world interactions, per hackathon rules [1] [2].

Features

- **URL/Endpoint Input:** A simple Next.js UI to enter a web resource (HTTP(S) URL, GraphQL endpoint, etc.).
- **Protocol Inspection:** The agent retrieves the resource (using real tools via Docker MCP) and analyzes its protocol behavior:
- **HTTP(S) Compliance:** Checks mandatory headers and values (HSTS, CORS, Content-Security-Policy, caching, cookies, etc.) against RFC references. For example, verifies that `Strict-Transport-Security` includes a valid `max-age` directive (e.g. “`max-age=31536000; includeSubDomains`” as shown on MDN [3]) and flags missing or malformed values.
- **GraphQL Validation (optional):** If a GraphQL endpoint is detected, introspects the schema (via an MCP GraphQL tool or code library) and validates a sample query’s syntax and type-safety per the GraphQL spec.
- **WebSocket Handshake (optional):** For WebSocket URLs, initiates a handshake and checks the `Sec-WebSocket-*` headers against RFC 6455.
- **Email/RFC Checks (future):** As an extension, could validate email address formats (RFC 5322) or OAuth flows by simulating an OAuth dance with known specs.
- **Detailed Report:** Returns a structured analysis highlighting which parts pass or fail (e.g. “HSTS header missing”, “CSP directive incorrect”, “CORS origin wildcard allowed – ensure correct header per spec”, etc.).
- **MCP Integration:** Uses at least one Dockerized tool (“MCP server”) inside the E2B sandbox to assist. For example, **Browserbase MCP** (a headless browser automation container) to fetch pages and extract text/content [4]. This ensures the agent works with real-world tools as mandated [1] [2].
- **(Optional) LLM Support:** The agent’s reasoning and reporting can be powered by a local LLM (e.g. via E2B Code Interpreter with GPT-4o) or even a Groq-accelerated model, adding a smart interpretation layer. Groq could be used to run an open model (e.g. Llama) locally for fast feedback, but this is optional since the core value is protocol parsing, not heavy LLM inference.

MCP Tools and How They Are Used

- **Browserbase MCP (Docker container):** We will integrate Browserbase's MCP server inside the E2B sandbox to programmatically browse to URLs and extract page content [4]. The agent uses `browserbase_stagehand_navigate` to load the target page, and `browserbase_stagehand_extract` to pull out text and metadata. (This satisfies the "use at least one Docker MCP" requirement [5].) Although Browserbase is a full browser, we mainly use it for reliable HTTP(S) requests and content extraction in the sandboxed environment.
- **Node.js HTTP client (inside sandbox):** In addition to Browserbase, the agent can use built-in code (via E2B's sandboxed runtime) to make direct HTTP requests (e.g. using `fetch` or `axios`) to retrieve raw headers and status codes. This allows precise header-level inspection (since Browserbase's extract tool strips non-text elements).
- **Groq (optional):** If time permits, we may use a Groq-supported LLM (e.g. Llama 3 via E2B) to perform complex parsing or reasoning about ambiguous protocol cases. For example, running a local model on Groq for quick compliance reasoning. However, this is ancillary and only included if it can be shoehorned with minimal friction (the main scoring focuses on the core idea and MCP usage).
- **Additional MCPs (if fitting):** We could optionally integrate other Docker MCPs like the official GitHub MCP (for pulling RFC documents or examples from a repo) or even Perplexity (to look up spec references). But the minimal plan is to use Browserbase; judges only require **≥1** MCP [5].

The overall setup uses E2B's new MCP Gateway: as E2B's blog explains, *"each MCP tool runs as a Docker container inside the E2B sandbox"* and they are accessed through a unified interface [2]. We will configure the E2B sandbox (via the E2B SDK) with our Browserbase API credentials so the agent code can call these tools. For example, E2B's code might do:

```
const sandbox = await Sandbox.create({
  mcp: { browserbase: { apiKey: env.BROWSERBASE_API_KEY, projectId:
env.BROWSERBASE_PROJECT_ID, geminiApiKey: env.GEMINI_API_KEY } }
});
// Now inside this sandbox, MCP calls route through Browserbase container.
```

This matches the recommended pattern [6] [2]. The agent can then run sequences like `await mcpClient.call('browserbase_stagehand_navigate', { url })` to load a page, and afterward `browserbase_stagehand_extract` to get content.

Technical Architecture and RFC Validation Integration

1. **Front-End (Next.js + shadcn UI):** A single-page interface where the user enters the target (URL or API) and sees results. The UI sends a request to our backend when "Run Checks" is clicked.
2. **API Layer (Next.js API Route):** Upon request, the API route invokes the E2B backend logic. This could use the E2B SDK or code-interpreter in TypeScript. For example, we might use `@e2b/code-interpreter` or the MCP client library (e.g. `mcp-use`) within this route. The API passes the user input to the agent logic.
3. **E2B Sandbox (secure agent runtime):** The core analysis happens here. E2B provides a secure sandbox (powered by Firecracker) where we run code generated or orchestrated by the LLM. The sandbox has the Browserbase MCP container already attached (via the MCP gateway) [2].

4. **LLM and Logic:** We use a Language Model (e.g. GPT-4o) inside the sandbox to formulate steps: fetch the resource, parse its response, and apply RFC rules. The LLM can be used in a code-interpreter mode, writing and executing code in the sandbox. For instance, the LLM might generate JavaScript to `fetch(url)` and then analyze headers. E2B's code interpreter will run this code and capture outputs.
5. **Tool Integration (MCP Gateway):** The agent's code can explicitly call the Browserbase MCP as a tool. This means the agent (or the orchestrating code) can do something like `const pageText = await client.mcp.browserbase.navigate(url)`, leveraging the Dockerized Browserbase tool. The retrieved data is then fed back to the agent for analysis.
6. **RFC/Spec Checks:** Once the data (headers/body) is obtained, the agent logic applies compliance rules. For example:
7. It checks if `Strict-Transport-Security` is present and correctly formatted (MDN notes a typical header as *"Strict-Transport-Security: max-age=31536000; includeSubDomains"* [3]). If missing or malformed, the agent flags an error.
8. It verifies other security headers (e.g. `X-Content-Type-Options: nosniff`, `X-Frame-Options`, `Content-Security-Policy`) and notes if they meet best practices (per OWASP/MDN guidance).
9. For CORS, it ensures `Access-Control-Allow-Origin` is not overly permissive unless intended.
10. (If GraphQL) It sends an introspection query and checks that a given GraphQL query is valid and follows schema rules (this uses a GraphQL library inside the sandbox, adhering to the official spec [7]).
11. (If WebSocket) It opens a WebSocket client inside the sandbox to the endpoint and confirms the handshake headers and frame format match RFC 6455.
12. **Result Aggregation:** The sandbox returns a structured report (JSON) with checks and suggestions. The Next.js API route receives this, then passes it back to the UI for display.

All sandbox code runs isolated, so even untrusted user input (a malicious URL or query) cannot escape the enclave. This demonstrates "trusted agent code execution with real tools" as the event intends [2] [1].

Code Plan and File Layout

- `/pages/index.tsx`**:** Main UI page (Next.js) with an input field and "Run Compliance Check" button. Uses *shadcn* components for form and results display.
- `/pages/api/check.ts`**:** API route receiving input (e.g. JSON `{ url: "https://example.com" }`). This will call our agent logic.
- `/lib/e2bAgent.ts`**:** Contains helper functions to interact with E2B. For example, a function `runComplianceCheck(input)` that:
- Creates or connects to an E2B sandbox (using the E2B SDK).
- Ensures Browserbase (and any other MCP) is attached.
- Sends instructions or code to the sandbox's LLM (e.g. via E2B Code Interpreter or MCP client). For instance, it might use a loop of LangChain-style prompt/response: *"Fetch the URL, extract headers, then check each header per RFC7230"*.
- Retrieves the result JSON from the sandbox.
- `/components/ResultCard.tsx` **(optional):** UI component to render individual check results (status, message).
- `/utils/complianceRules.ts`**:** (Optionally) A list of rules or a function library for specific checks (e.g. regex for header formats). This can assist the agent or be called inside the sandbox.

- `/styles/` **etc:** Any CSS or theme as needed by shadcn.

Data Flow: User submits → API calls `runComplianceCheck(input)` in E2B → E2B sandbox code runs tool calls (Browserbase, fetch, parsing) → returns analysis JSON → API responds → UI displays results.

By structuring the code this way, we clearly separate the front-end from the agent logic. The heavy lifting (network calls, parsing) is done in the sandbox via safe, containerized tools. We would also include error handling (e.g. if the target URL is unreachable) to make the demo smooth.

Hour-by-Hour Timeline (24h)

We break the 24h hackathon into focused blocks to ensure completion:

- **Hour 0–2:** *Project setup.* Scaffold a new Next.js TypeScript project. Add shadcn-ui components. Set up GitHub repo and initial commit. Create basic UI (`index.tsx`) with an input field and button.
- **Hour 2–4:** *E2B & MCP integration.* Install E2B SDK (`npm install e2b`) and obtain API keys for Browserbase (and any MCP). Write helper code to create an E2B sandbox with the Browserbase MCP attached. Test connectivity (e.g. call a simple MCP command like `browserbase_stagehand_navigate` on a known URL to ensure the tool is reachable). [6] [4]
- **Hour 4–7:** *Core agent logic – HTTP fetch and header extraction.* In the sandbox code, implement functionality (via Node or Browserbase) to fetch the given URL and retrieve HTTP response headers. Verify that the basic chain works end-to-end: user input triggers sandbox run, and headers are returned.
- **Hour 7–10:** *Protocol compliance checks.* Implement the rule checks in the sandbox. For each important header (HSTS, CSP, CORS, etc.), write code (or prompt the LLM) to validate against RFC expectations. For example, check if `Strict-Transport-Security` exists and has a numeric `max-age` (per MDN example [3]). Log pass/fail for each. Test on several sites (e.g. example.com, a site with missing headers, etc.) to refine logic.
- **Hour 10–12:** *GraphQL/WebSocket (stretch).* If time permits, implement an optional mode: detect if the URL ends in `/graphql` or query param, then use a GraphQL library (or `mcp/text-to-graphql` MCP) to fetch the schema and validate a sample query. Also try WebSocket: open a `WebSocket` from within sandbox and see if handshake succeeds. If too complex, skip this and focus on core HTTP.
- **Hour 12–15:** *UI polishing and wiring results.* In Next.js, display the compliance report in a user-friendly way (e.g. color-coded pass/fail list). Add any shadcn components for styling. Test the full flow: UI → API → sandbox → results. Make the output concise and visually clear for a demo.
- **Hour 15–18:** *Error handling and robustness.* Handle cases like invalid URLs, timeouts, or tools failing. Ensure sandbox errors are caught and shown. Add logging in the demo (optionally show partial sandbox output).
- **Hour 18–20:** *Add a demo dataset.* Prepare a few example URLs or endpoints to showcase (some that are compliant, others intentionally missing headers). Possibly create a small example page/server (within E2B or as a static file) that deliberately violates an RFC for demonstration.
- **Hour 20–22:** *Documentation & Presentation prep.* Write a brief README, comments in code. Plan demo script: e.g. first show a compliant site, then one with errors. Rehearse explaining the tech stack and how MCPs are used securely.

- **Hour 22–24:** *Final testing and submission.* Record the 2-minute demo video (as required) showing the UI, agent in action, highlighting the E2B sandbox usage (maybe showing logs or explanation overlay). Submit before the 9:00 PDT deadline.

Throughout, continuously commit code to Git and push to GitHub (to have a full working submission). Prioritize having a clear, working minimum (HTTP checks only) first, then add any extras if time allows. The schedule emphasizes **technical depth** early (getting the agent fetching and checking headers) before UI cosmetic touches, which matches the judging emphasis on innovation and technical quality [8].

Demo Best Practices

- **Show E2B Sandbox in Action:** In the demo, briefly explain that all code is running in an isolated E2B environment. If possible, show a quick view (or mention) of the sandbox creation log or the MCP gateway (the E2B dashboard or logs) to reinforce the "sandboxed agent" aspect [2].
- **Highlight MCP Use:** Demonstrate a step where the agent calls the Browserbase MCP. For instance, explain "the agent is using Browserbase to fetch this page content" while maybe showing a console log output from that tool. This directly addresses the hack requirement of "real tools from Docker Hub" [5].
- **Protocol Focus:** When presenting results, explicitly reference the specs. E.g. "Here the HSTS header is missing (`max-age` absent), which violates RFC 6797 [3]. Our agent catches that automatically." Use concrete terms like "RFC" and "spec" to show protocol literacy.
- **Innovation Points:** Emphasize anything novel (e.g. "Unlike a generic web agent, CompliBot parses raw protocol frames and checks against official standards"). If you added Groq or advanced LLM steps, mention how that speeds analysis or adds intelligence (but note it was optional).
- **Keep it Concise:** Since demo time is limited, prepare a short script. Start with a compliant example to build confidence, then a faulty example to show agent usefulness. End with a summary of architecture (perhaps a quick slide or diagram).
- **Prepare for Questions:** Expect judges to ask how you used Docker MCPs and E2B. Have the key lines of code ready (e.g. showing the E2B sandbox creation with MCP config) and the event citations ("We used Browserbase MCP within E2B sandbox as required [5] [2]").

By following this plan, CompliBot will present a well-architected, technically deep solution that clearly meets the hackathon's rules (E2B sandbox + Docker MCP) and focus area (protocol compliance). The thoroughness of compliance checks and the correct use of the E2B+MCP stack align with the judging criteria of technical quality and innovation [8] [2].

References: Official hackathon rules and tips [1] [5]; Docker/E2B MCP partnership blog [2] [9]; Browserbase MCP documentation [4]; and MDN on HTTP headers [3]. These guided the project design and tool usage.

---

[1] [5] [8] Build MCP Agents - with Docker, Groq, and E2B · Luma
https://luma.com/0vm36r4q

[2] [6] [9] Docker & E2B partner to introduce MCP support in E2B Sandbox — E2B Blog
https://e2b.dev/blog/docker-e2b-partner-to-introduce-mcp-support-in-e2b-sandbox

3 Strict-Transport-Security header - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Strict-Transport-Security

4 Browserbase | Model Context Protocol (MCP) Integration
https://www.browserbase.com/mcp

7 GraphQL Specification
https://spec.graphql.org/October2021/