Blueprint for a 48-hour multi-agent simulation platform

The fastest path to an impressive hackathon demo combines Next.js with either the Vercel AI SDK (TypeScript) or OpenAI Agents SDK + FastAPI (Python), MapLibre + deck.gl for visualization, and pre-loaded data supplemented by a few live APIs. The critical insight: pre-write all agent prompts, pre-load all geographic data, and deploy a skeleton app in the first two hours. MCP is feasible but optional — direct tool definitions get you there faster. The map is your visual anchor; the streaming multi-agent analysis is your technical wow factor.

DevSwarm is not what you think it is

There are two products called "DevSwarm," and neither is a scaffolding tool in the traditional sense.

DevSwarm by devswarm.ai is a desktop application (GitHub) (Mac/Windows) created by Twenty Ideas, an Oregon-based agency. It launched in beta in September 2025. Rather than generating code from prompts, it's a **parallel AI orchestration environment** — you run multiple AI coding assistants (Devswarm) (Claude Code, Gemini, Codex, Copilot, etc.) simultaneously on isolated Git branches called "Builders." (devswarm) Each Builder gets its own worktree, (Devswarm) preventing merge conflicts. (GitHub) You can run **10+ agents in parallel** on the free tier, compare their outputs, and merge the best branches. (Intellyx) It supports Jira integration, (Devswarm) GitHub PR management, and MCP integrations for testing (Playwright) and design (Figma). (Devswarm) Pricing is free for non-commercial use, $20/month for Pro. (devswarm)

DevSwarm by The-Swarm-Corporation is a small Python library (8 GitHub stars, 9 commits) (github) that combines the Swarms framework with v0.dev for code generation from prompts. It's too immature and risky for a hackathon.

For your hackathon, the practical recommendation: Use **Bolt.new** or **v0.dev** to scaffold the initial project in minutes, then export to GitHub and develop in **Cursor** or **DevSwarm (devswarm.ai)** for parallel feature development. DevSwarm's real value is letting team members run multiple AI assistants on different features simultaneously (Devswarm) — one Builder on the map component, another on agent orchestration, a third on the API layer. (devswarm) The common emerging pattern is: scaffold with Bolt.new → iterate with Cursor/DevSwarm. (Imagine)

OpenAI Agents SDK wins for hackathon agent orchestration

The Responses API is OpenAI's new foundation

The Responses API launched in March 2025 as the successor to Chat Completions. (Medium) It's now OpenAI's **recommended API for all new projects**. (OpenAI) Key advantages include built-in tools (web search, file search, code interpreter, computer use, MCP server integration), stateful context via (previous_response_id)

chaining, OpenAI **40–80% improved cache utilization**, OpenAI and background mode for long-running operations. The Assistants API sunsets August 26, 2026, with all features migrating to Responses. OpenTools.ai

Current pricing per million tokens:

Model	Input	Cached Input	Output
GPT-4.1	$2.00	$0.50	$8.00
GPT-4.1-mini	$0.40	$0.10	$1.60
GPT-4o	$2.50	$1.25	$10.00
GPT-4o-mini	$0.15	$0.075	$0.60

The Agents SDK is the fastest multi-agent framework

The **OpenAI Agents SDK** is a lightweight, open-source Python framework built on the Responses API but provider-agnostic (supports 100+ LLMs via LiteLLM). DigitalOcean Its core primitives — Agents, Handoffs, Guardrails, and Sessions — are minimal yet powerful. The key pattern for your project is "**Agents as Tools**": wrap each specialist agent as a `@function_tool`, then let an orchestrator call all five in parallel via `parallel_tool_calls=True`.

```
User Query → Orchestrator Agent (GPT-4.1)
  ├── Geopolitics Agent (GPT-4.1-mini)   [parallel]
  ├── Economy Agent (GPT-4.1-mini)       [parallel]
  ├── Food Supply Agent (GPT-4.1-mini)   [parallel]
  ├── Infrastructure Agent (GPT-4.1-mini) [parallel]
  └── Civilian Impact Agent (GPT-4.1-mini)[parallel]
→ Synthesis Agent (GPT-4.1)
→ Structured output → Map visualization
```

Estimated cost per scenario: $0.05–0.10 without web search, $0.20–0.35 with search. Total hackathon budget (200 dev runs + 50 demo runs): **$50–100**.

How it compares to alternatives

Factor	OpenAI Agents SDK	CrewAI	LangGraph	AutoGen
Setup time	Minutes	~30 min	Hours	Hours
Learning curve	Low	Low-Medium	High	Medium

Factor	OpenAI Agents SDK	CrewAI	LangGraph	AutoGen
Parallel agents	Native asyncio	Supported	Graph-native	Supported
Built-in tracing	Yes	Limited	Via LangSmith	Manual
Hackathon fit	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐

CrewAI is the runner-up — its role/task/crew metaphor maps naturally to domain specialists, but debugging is harder. Sider **LangGraph** is powerful for complex conditional workflows but overkill for 48 hours unless the team already knows it. **AutoGen** has confusing versioning and steeper setup.

The **Vercel AI SDK** (TypeScript) is equally viable if the team prefers a full-stack TypeScript approach — it offers an `Agent` class with built-in SSE streaming, subagent patterns, and React hooks (`useChat`) for consuming streams. Vercel The choice between OpenAI Agents SDK (Python) and Vercel AI SDK (TypeScript) depends entirely on team skill composition.

MCP is surprisingly feasible and worth building

The Model Context Protocol has matured significantly since Anthropic open-sourced it in November 2024. Axway The latest spec (version 2025-11-25) supports tasks, parallel tool calls, and server-side agent loops. Modelcontextprotocol **Every major provider now supports MCP**: OpenAI's Agents SDK has native `MCPServerStdio` and `MCPServerStreamableHttp` classes, OpenAI Claude has built-in support, and VS Code/Copilot works with MCP servers. DigitalOcean

Building a custom MCP server with **FastMCP** (now part of the official Python SDK) takes **2–4 hours**:

```python
```

```python
from mcp.server.fastmcp import FastMCP
import httpx

mcp = FastMCP("scenario-data")

@mcp.tool()
async def get_geopolitical_events(region: str, days: int = 7) -> str:
    """Query GDELT for recent geopolitical events in a region"""
    async with httpx.AsyncClient() as client:
        resp = await client.get(f"https://api.gdeltproject.org/api/v2/doc/doc?query={region}&timespan={days}d&format=json")
        return resp.text

@mcp.tool()
async def get_economic_indicators(country_code: str) -> str:
    """Get key economic indicators from World Bank"""
    # ... World Bank API call
```

The critical design principle: **create task-level tools, not 1:1 API wrappers**. A single `analyze_food_security(region)` tool that internally queries FAO, World Bank, and NASA POWER is far more useful to an agent than three separate API tools.

However, for a 48-hour hackathon, MCP is a "nice to have" rather than essential. Direct function/tool definitions passed to the OpenAI Agents SDK or Vercel AI SDK achieve the same result with less protocol overhead. Medium Build MCP if you want the demo value (judges will recognize it) or if you're using Claude Desktop for development. Skip it if time is tight — the agents work just as well with plain Python functions decorated as tools.

Data source API playbook: what works and what to skip

Tier 1: Integrate immediately (no auth, instant access)

GDELT Project is the crown jewel for real-time geopolitical data. The DOC 2.0 API (`api.gdeltproject.org/api/v2/doc/doc`) requires no authentication, returns JSON, and searches a rolling 3-month window of global news GDELT Project with 15-minute update granularity. The GEO 2.0 API returns GeoJSON with geolocated event data GDELT Project — perfect for map overlays. BigQuery access provides the full historical dataset SitePoint (back to 1979) with **1 TB free querying per month**. SitePoint Python libraries `gdeltdoc` and `gdeltPyR` simplify access. GitHub Caveat: data is auto-coded from news media, so expect noise PyPI and English-language bias.

World Bank Open Data API (`api.worldbank.org/v2/`) is completely open — no auth, no documented rate limits. It covers **16,000+ indicators** across all countries: GDP, population, poverty, unemployment, inflation,

trade, arable land, energy use. Data is primarily annual, so pre-load everything at startup and serve from cache. Key indicators for your scenario: NY.GDP.MKTP.CD (GDP), SI.POV.DDAY (poverty), AG.LND.ARBL.ZS (arable land), EG.USE.PCAP.KG.OE (energy use).

UNHCR Refugee Data Finder (api.unhcr.org/population/v1/) requires no authentication and provides **70+ years of displacement data** — refugees, asylum-seekers, IDPs, and stateless persons by country. JSON output with pagination. Ideal for displacement modeling.

NASA POWER API (power.larc.nasa.gov/api/) provides solar, temperature, wind, and precipitation data optimized for energy and agriculture analysis. Free API key from api.nasa.gov (instant), **1,000 requests/hour**. Excellent for location-specific climate impact data.

Tier 2: Register early, integrate on Day 1

ACLED (Armed Conflict Location & Event Data) is the gold standard for conflict event data — weekly-updated, geolocated events with actors, fatalities, and event types. Requires registration at developer.acleddata.com; **approval may take hours to days**. Register immediately and use GDELT as a fallback.

INFORM Global Risk Index has no API — download the Excel file from the EU JRC DRMKC site, parse it, and pre-load. Covers **191 countries** with risk scores across three dimensions: Hazard & Exposure, Vulnerability, Lack of Coping Capacity. Our World in Data Updated twice yearly. Unescap

Copernicus Climate Data Store requires free registration and has queue-based processing that **can take hours**. Pre-download ERA5 climate summaries and CMIP6 scenario projections before the hackathon. Do not rely on live CDS queries during a demo.

Tier 3: Skip or pre-load as static data

Stanford HAI AI Index has no API — it's a PDF report with downloadable datasets on Kaggle. Pre-load key metrics as static reference data. **SIPRI** (arms transfers, military expenditure) is download-only Excel/CSV. **FAO/FAOSTAT** has a REST API but adds marginal value beyond World Bank data for a demo. **OpenStreetMap Overpass API** is powerful for infrastructure queries but rate-limited on the public server — pre-query and cache infrastructure layers for target regions.

The most valuable additional source: The **WRI Global Power Plant Database** is a single CSV with ~35,000 power plants worldwide (name, capacity, fuel type, lat/lon). Download once, load instantly. Perfect for infrastructure vulnerability visualization.

Recommended 48-hour data strategy

Pre-load everything possible before the hackathon: INFORM Risk Excel, WRI Power Plants CSV, World Bank indicators for all countries, Natural Earth GeoJSON boundaries. During the hackathon, integrate GDELT and UNHCR APIs live (they're the easiest), and use pre-cached data for everything else. **GDELT is the only source that truly benefits from real-time access** Gdeltproject (15-minute updates).

deck.gl + MapLibre is the winning map stack

Why deck.gl dominates for scenario visualization

deck.gl (originally by Uber, now maintained by vis.gl) is a WebGL2/WebGPU-powered framework whose layer catalog maps directly to your visualization needs:

- **GeoJsonLayer**: Choropleth maps for economic impact by country, (deck.gl) with data-driven fill colors and 3D extruded polygons for severity
- **HeatmapLayer**: GPU-accelerated Gaussian kernel density (deck.gl) for conflict intensity or climate risk
- **ArcLayer**: Trade route disruptions, migration flows, supply chain connections between regions
- **IconLayer**: Critical infrastructure markers, conflict hotspots, refugee camps
- **TripsLayer**: Animated displacement paths or supply chain disruptions over time
- **ContourLayer**: Risk threshold isolines (e.g., "high risk" / "extreme risk" zones)

deck.gl's [transitions] prop enables **smooth animated color changes** as agents produce analysis — the visual effect of watching regions change color as cascading impacts propagate is the demo "wow factor."

Use MapLibre over Mapbox for zero cost

MapLibre GL JS is the open-source fork of Mapbox GL JS (BSD-2-Clause license, completely free). Combined with free tile providers (MapTiler free tier, Stadia Maps, or Protomaps), it provides equivalent visualization capabilities at **zero cost**. Use [react-map-gl/maplibre] for React integration — same API as the Mapbox version. deck.gl has first-class MapLibre integration via [@deck.gl/mapbox].

Mapbox GL JS offers **50,000 free map loads/month** (Storemapper +2) if you prefer higher-quality tiles, but MapLibre eliminates any billing risk during a demo. For a hackathon, MapLibre is the pragmatic choice.

Skip Leaflet and Kepler.gl

Leaflet is raster-based, not WebGL — it can't handle complex animated overlays, multiple dynamic layers, or smooth real-time transitions. It's a fallback option only. **Kepler.gl** can be embedded as a React/Redux component, but its heavy dependency footprint (Redux, react-palm, Styled-Components), opinionated UI, and poor documentation (OddBlogger) make it a time sink for a 48-hour build.

Real-time agent updates to the map

Use **Server-Sent Events (SSE)**, not WebSockets. Agent output streaming is fundamentally one-directional (server → client), SSE has built-in automatic reconnection, and Vercel's serverless architecture doesn't support persistent WebSocket connections. The pattern:

1. Backend streams agent analysis as SSE events: `data: {"agent": "geopolitics", "region": "SYR", "impact_score": 8.7}`
2. React state updates per-agent using isolated components to prevent cascade re-renders
3. deck.gl layers automatically re-render with smooth transitions when data props change
4. Side panel shows detailed agent analysis text for the clicked region

The interactive region pattern

Full-width dark-themed map (80% of viewport) with a collapsible right sidebar (20%). Set `pickable: true` on deck.gl layers. On click, the sidebar slides in showing the region name, impact scores from each agent, streaming agent analysis text, and mini charts (via Recharts) for time-series data. Map highlights the selected region with enhanced borders. This layout is standard for geospatial dashboards and immediately communicates professionalism.

The two viable tech stacks for your 48 hours

Option A: Full TypeScript (recommended for most teams)

```
FRONTEND + BACKEND (single Next.js 15 app)
├── Vercel AI SDK 6 (Agent class, streamText, SSE streaming)
├── react-map-gl + MapLibre GL JS + deck.gl
├── shadcn/ui + Tailwind CSS
├── Recharts for sidebar mini-charts
└── Deployed to Vercel (single git push)

AGENT ORCHESTRATION (Next.js API routes)
├── 5 specialist agents + 1 synthesis agent
├── Fan-out parallel execution via Promise.allSettled()
├── Structured outputs via Zod schemas
└── AI Gateway for multi-provider fallback
```

Why this wins: Single deployment, single language, SSE streaming built in, `useChat` hooks for consuming streams, Vercel deploy in seconds. The Vercel AI SDK handles agent loops, tool execution, and streaming automatically. `Vercel` No CORS, no Docker, no multi-service debugging at 3am.

Option B: Python backend + TypeScript frontend (for Python-heavy teams)

```
FRONTEND (Next.js on Vercel)
├── react-map-gl + MapLibre + deck.gl
├── shadcn/ui + Tailwind CSS
```

```
└── EventSource consumption of SSE streams

BACKEND (FastAPI on Railway)
├── OpenAI Agents SDK (agents-as-tools pattern)
├── FastMCP server wrapping data APIs
├── asyncio.gather() for parallel agent execution
├── StreamingResponse for SSE
└── Pre-loaded data in memory
```

Why this works: Python is the lingua franca of AI/ML, the OpenAI Agents SDK is the simplest multi-agent framework available, and FastAPI's async support handles parallel agent execution natively. The tradeoff is two deployments and CORS configuration.

The deciding factor is your team's skill composition. If everyone writes TypeScript, go Option A. If the team is Python-strong with one React developer, go Option B.

What to build, what to fake, and how to win

The realistic 48-hour scope

Build live: the agent orchestration pipeline (it's the core differentiator), map visualization with pre-defined choropleth layers, streaming agent output to the UI, a scenario input form with 3 pre-tested "golden path" scenarios, and click-to-detail region interactions.

Pre-load or mock: all geographic data (Natural Earth GeoJSON, WRI power plants, INFORM risk scores, World Bank indicators), all agent system prompts (write and test these *before* the hackathon), map tile configuration, and example scenario parameters. **Do not build**: user authentication, persistent storage, mobile responsiveness, custom-trained models, or real-time ingestion from slow APIs (Copernicus).

The three highest-risk components

Map integration complexity is the number one risk. deck.gl + react-map-gl setup can be finicky with tile loading, layer rendering, and camera synchronization. Assign one dedicated person as the map specialist and have a Leaflet fallback ready. **LLM API rate limits and latency** could break the demo if five parallel calls timeout — implement response caching for repeated scenarios and have pre-cached "golden path" responses as a safety net. **Streaming state management** with five parallel agent streams updating React simultaneously can cause performance issues — isolate each agent's output into its own component and throttle state updates.

The three-minute demo that wins

1. App opens to a beautiful dark-themed world map filling the screen
2. Select a pre-built scenario: "*Suez Canal blockage during extreme heat event in South Asia*"

3. Click "Analyze" — map zooms to the affected region
4. Five agent panels appear on the sidebar, each **streaming analysis token-by-token** in real time
5. Map overlays animate simultaneously: trade routes turn red (ArcLayer), climate risk zones pulse (HeatmapLayer), infrastructure stress points appear (IconLayer), displacement paths animate (TripsLayer)
6. A synthesis panel appears: "*Here's how these cascading impacts interact across domains...*"
7. Click on India — sidebar shows country-specific impacts across all five dimensions with mini charts
8. Modify a parameter (escalation level) → agents re-analyze with visible changes

This impresses Snowflake engineering judges because: it demonstrates real-time multi-agent orchestration, streaming data pipelines, geospatial visualization, and structured data extraction from unstructured AI analysis — all themes that resonate with a data infrastructure company. Frame the narrative as "turning unstructured scenario data into structured, actionable geospatial intelligence."

The 48-hour timeline

Hours 0–6 (Foundation): Align team, assign roles, deploy skeleton Next.js app to Vercel, install dependencies (AI SDK or Agents SDK, react-map-gl, deck.gl, shadcn/ui), render a basic map with world GeoJSON, build page layout with sidebar.

Hours 6–20 (Core features): Define and test all 5 agent system prompts (parallel track), implement agent orchestration with parallel fan-out and synthesis, connect streaming output to the UI, build choropleth + heatmap layers with pre-loaded data, implement scenario input form, wire click-to-detail interaction.

Hours 20–32 (Integration and polish): End-to-end testing of all 3 golden-path scenarios, animated map overlays (arcs for trade routes, trips for displacement), UI polish (loading states, transitions, dark theme refinement), fix the inevitable streaming bugs.

Hours 32–40 (Demo prep): Feature freeze at hour 30. Record backup demo video at hour 40. Rehearse the 3-minute demo script. Handle edge cases. Deploy final version.

Hours 40–48: Rest, final rehearsal, present.

Strategic pre-hackathon preparation

The biggest time savings come from work done *before* the 48 hours start. **Pre-write and test all five agent system prompts** — each should be 200–500 words of carefully crafted instructions with output format specifications (JSON schema for structured data that maps to map layers). **Pre-download all static data**: Natural Earth GeoJSON (110m resolution, ~600KB), WRI Global Power Plant Database (CSV), INFORM Risk Index (Excel), World Bank indicators for all countries (bulk query the API). **Pre-configure all API keys**: OpenAI/Anthropic, MapTiler (free tier for tiles), and optionally ACLED and Copernicus (register now, approval may take days). **Pre-build the map component** if hackathon rules allow — getting deck.gl + MapLibre rendering correctly is the single most time-consuming setup task.

Conclusion

The technical stack decision is straightforward: **Next.js + Vercel AI SDK + MapLibre + deck.gl** for TypeScript teams, or **Next.js + FastAPI + OpenAI Agents SDK** for Python teams. Both paths can deliver a working demo. The non-obvious insight is that the *preparation* matters more than the *technology* — pre-written agent prompts, pre-loaded GeoJSON data, and pre-tested golden-path scenarios are what separate winning hackathon projects from ambitious failures. MCP is worth building only if you have bandwidth after core features work; direct tool definitions achieve the same result faster. The map visualization, not the AI orchestration, is likely the component that will consume the most debugging time — staff it accordingly. And above all: deploy in hour 2, feature-freeze at hour 30, and never demo a scenario you haven't tested at least five times.