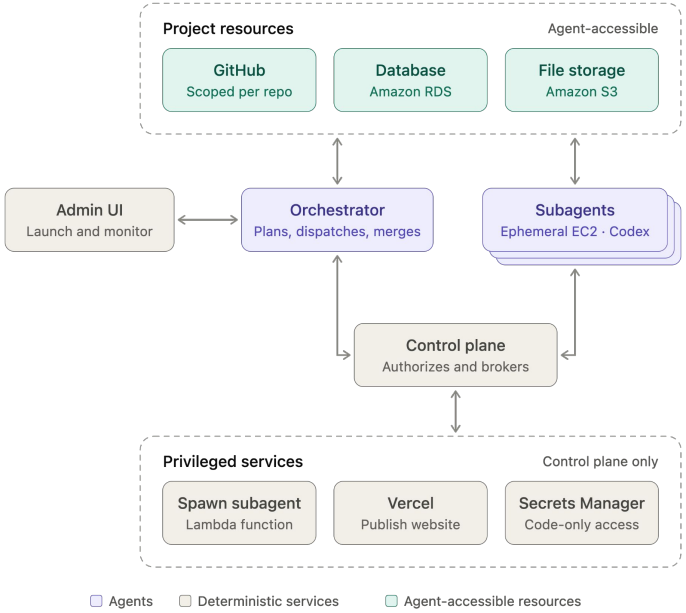
Building a Persistent Multi-Agent System

Project Walkthrough · Cody Comyns

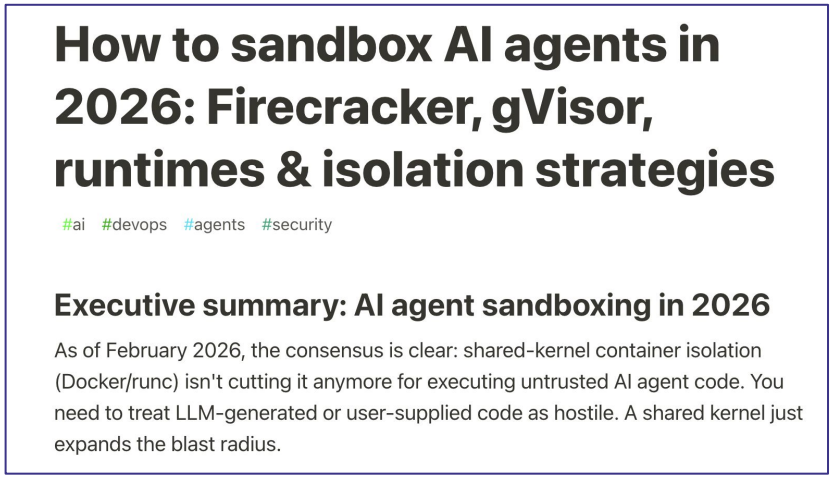
Part 1: High-Level Design

Overview



- Built a distributed multi-agent system with up to 13 concurrent Codex agents (1 orchestrator + 12 subagents), each running in an isolated EC2 instance, to execute open-ended research and data mining tasks
- Created a serverless control plane that enforces the 12 subagent concurrency limit and dynamically provisions scoped IAM policies, restricting each subagent to its assigned S3 prefix
- Implemented a persistent memory system by assigning each project a unique GitHub repository, S3 bucket, RDS database, Vercel Project, and ledger of markdown files explaining the current state of the project, allowing for multi-agent runs to be resumed

One VM, One Agent



How to sandbox AI agents in 2026: Firecracker, gVisor, runtimes & isolation strategies

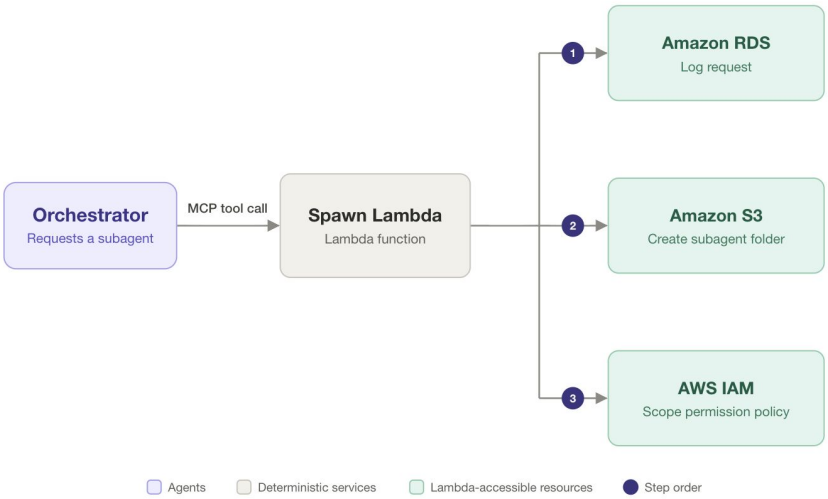
#ai #devops #agents #security

Executive summary: AI agent sandboxing in 2026

As of February 2026, the consensus is clear: shared-kernel container isolation (Docker/runc) isn't cutting it anymore for executing untrusted AI agent code. You need to treat LLM-generated or user-supplied code as hostile. A shared kernel just expands the blast radius.

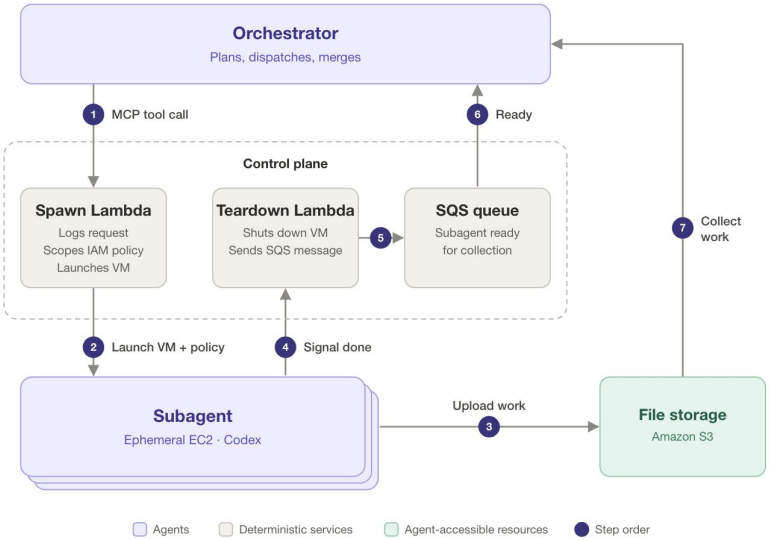
- A VM gives each agent its own kernel, so a kernel exploit can’t reach other agents
    - I used EC2 instances with prebaked AMIs, which reduces cold start time from dependency setup, but it’s still much slower than it should be (~1 minute)
- In hindsight, I would’ve used Modal’s sandboxes or a microVM service (AgentCore, E2B), which have <1 second cold start time

Orchestrator <> Subagent Communication



- The orchestrator cannot spawn subagents directly
- To spawn a subagent, the orchestrator calls an MCP tool that sends a request to a Lambda function
- The Lambda function logs the request, builds a scoped permission policy for the subagent, and launches its VM with that policy attached
- When finished, the subagent uploads its work to S3 and calls a Lambda function to signal it is done, which shuts down the subagent VM and sends an SQS message that the subagent is ready for collection

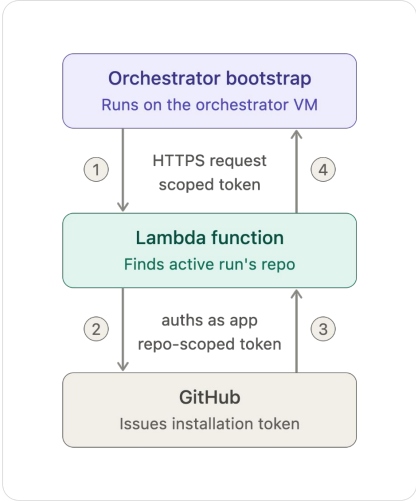
Subagent Collection



- The completion Lambda shuts down the subagent VM and sends an SQS message that the subagent is ready for collection
- An SQS listener in the orchestrator VM polls the queue continuously and picks up the message
- The listener uses OpenAI's turn/steer to insert a collection notice at the top of the orchestrator's context window
- I overrode the coding agent's initial design of putting the subagent id directly in the orchestrator's context window (to avoid weird tokenization errors, like counting the r's in strawberry) and instead used a local JSON registry in the orchestrator VM that maps each subagent collection to a new entry, marked with "status":"read"

GitHub Integration

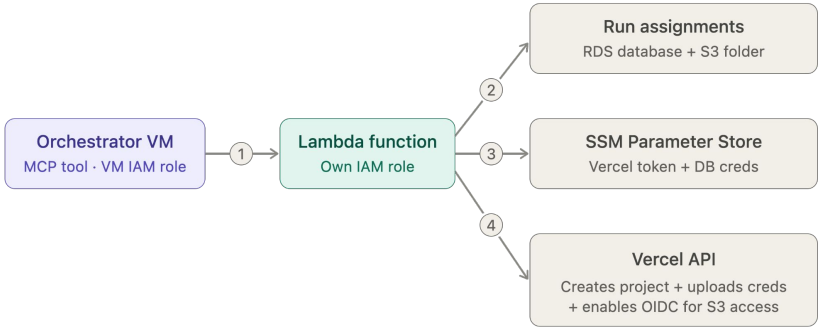
- Created a private GitHub organization to keep track of all my projects associated with this system
- Within that GitHub org, I created three GitHub Apps: (1) with permissions to create new repos, creds held by admin UI (2) with permissions to write to repos in the org, creds held by orchestrator VM, and (3) with permissions to launch new Vercel projects/deployments
- Right now, the orchestrator is the only agent with permission to write to the repo




```
{
  "repository_ids": [assignedRepo],
  "permissions": {
    "contents": "write",
    "metadata": "read"
  }
}
```

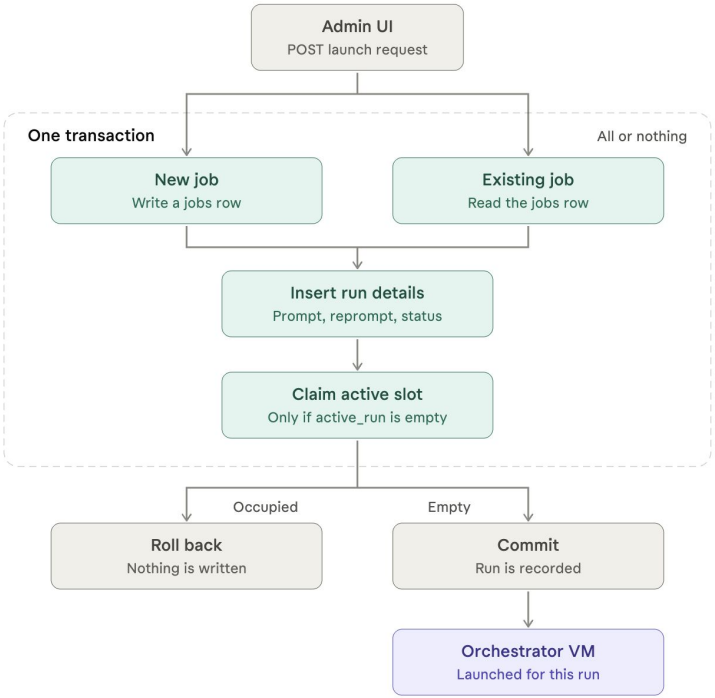
Vercel Integration

- New multi-agent project → new Vercel deployment needs to be created (sequential runs just reuse the existing Vercel project and git pushes get auto-synced)
- Process for new Vercel Deployment:
    - Orchestrator calls MCP tool → request to lambda function → lambda looks up run's repo and creates a new Vercel Project → lambda uploads the database url and the name of the s3 folder/prefix
    - For s3 specifically, requests from the frontend go through Vercel's OIDC to request a token with temporary s3 access



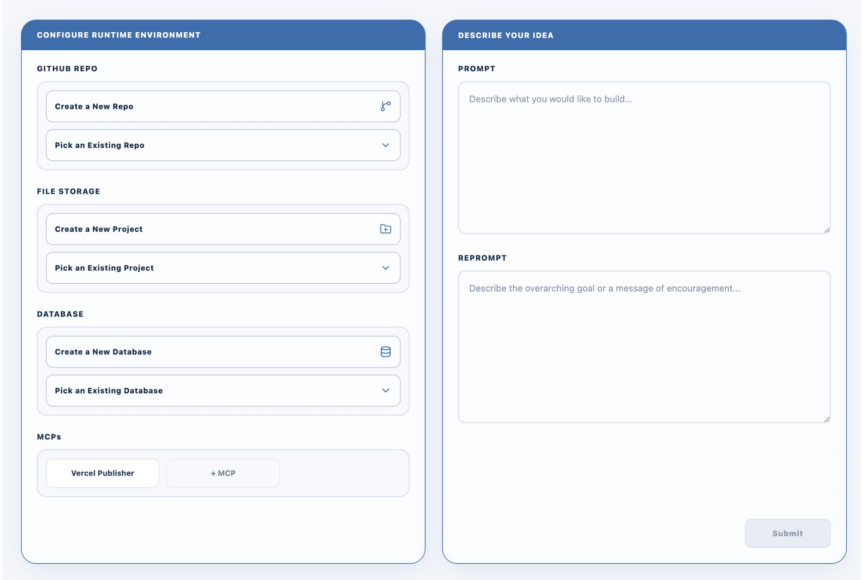
Part 2: Low-Level Design

Lifecycle of a Multi-Agent Run



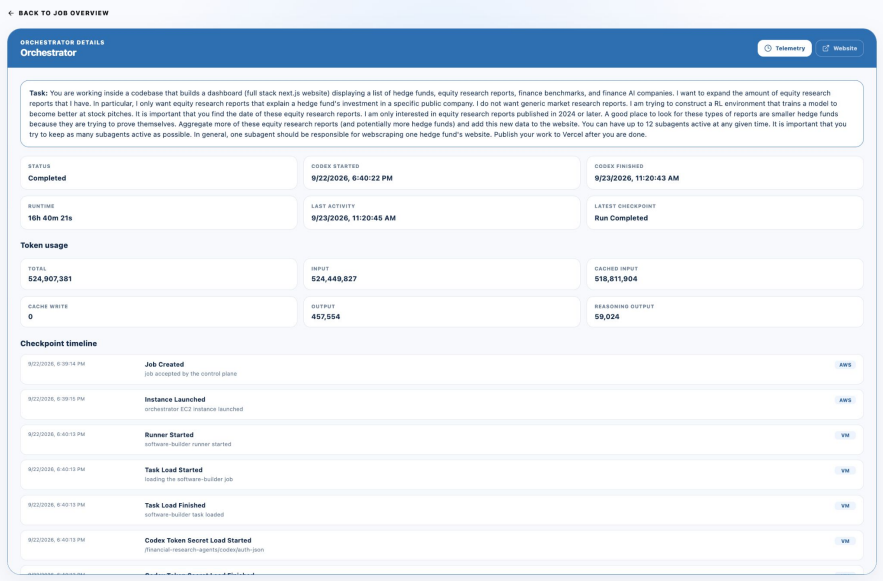
- Important terminology: a job can be composed of many multi-agent runs, and each job is assigned a unique set of cloud resources
- Upon clicking the “Launch Job” button, the admin UI sends a “POST /api/jobs” request
- Inputs are validated, and then the api handler decides which of two paths to take
  - Path #1: new job, first run
  - Path #2: existing job, follow-up run
- These two paths execute slightly different database transactions (path #1 executes a write operation in the jobs db table, whereas path #2 does not)
- Three total db tables: jobs, run_details, active_run (this last table functions like a lock with a conditional update)

Reprompt Loop



- On open-ended tasks, the orchestrator quits too early
  - Example: I tasked it with gathering the portfolio companies of 250 VCs and it stopped after 58 of them
- My solution: run the orchestrator in a while loop. Each time it stops, it's restarted with a reprompt layered on its prior context
  - This reprompt is only activated when the orchestrator stops AND there are no active subagents
- The system keeps running until the user explicitly clicks “END JOB” in the admin UI
- Motivation for this: tweet from Anthropic researcher working on the Riemann Hypothesis

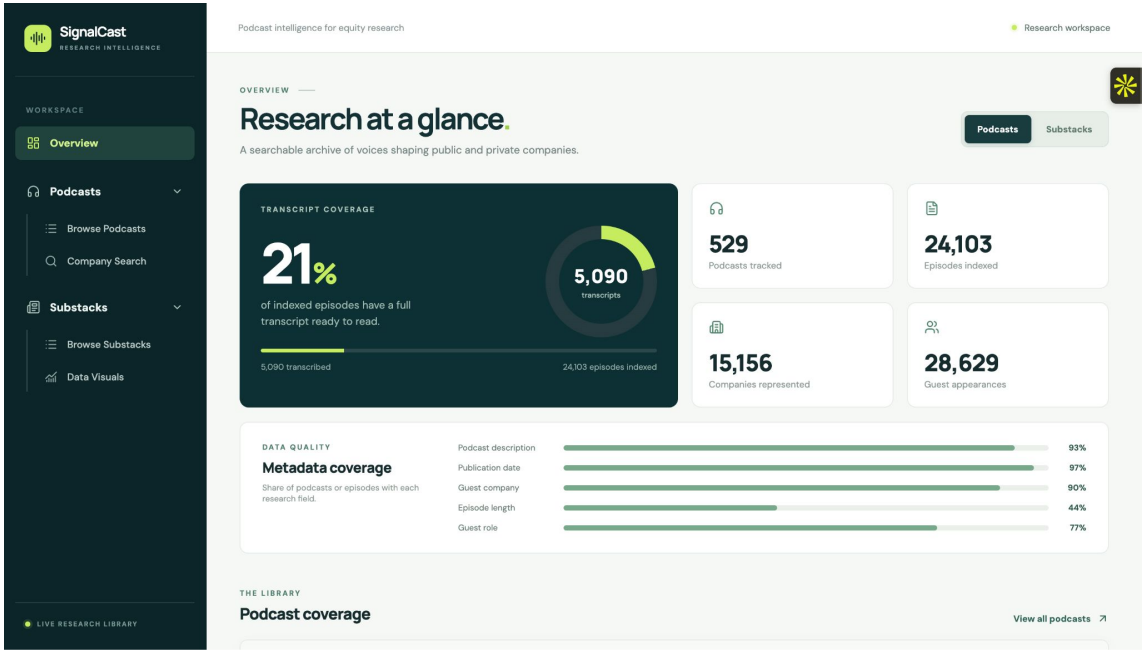
Agent Observability



- The agent runner script timestamps each checkpoint (ie. credential retrieval, orchestrator reprompt, end job, etc...) and logs it to s3
- Similarly, the agent runner script collects token usage data from OpenAI's SDK (which codex emits after each event, running tally style), which is split into input, cached input, output, and reasoning output
- Primarily use this observability data to see how efficiently the agents are working together (ie. if orchestrator keeps repeatedly launching subagents to investigate the same issue)

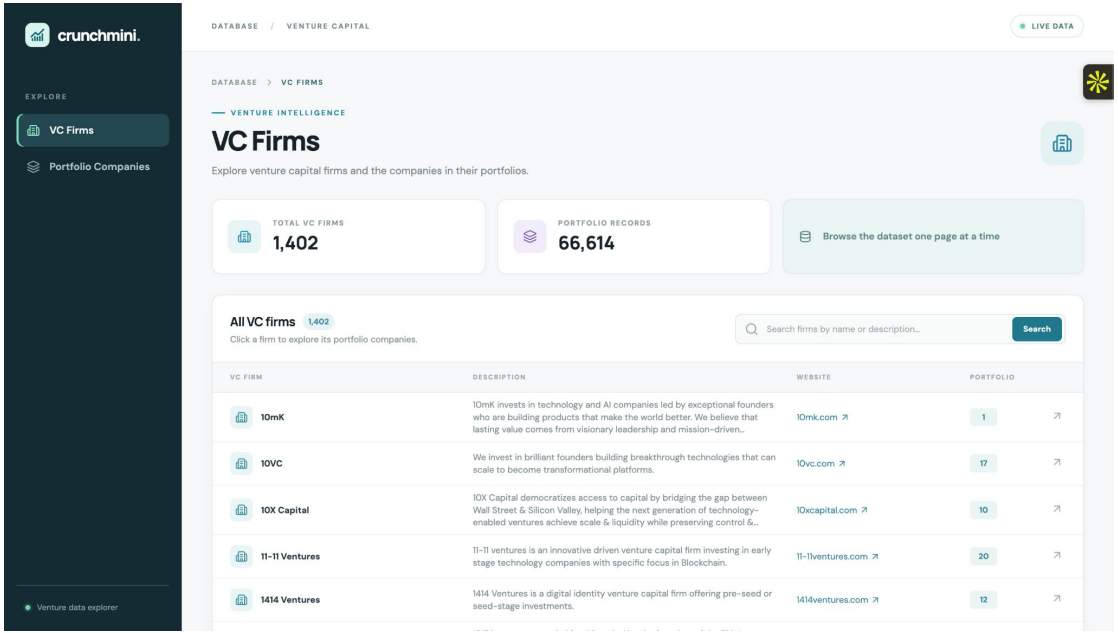
Part 3: Some Things I’ve Built with This System

Sample Project (1)



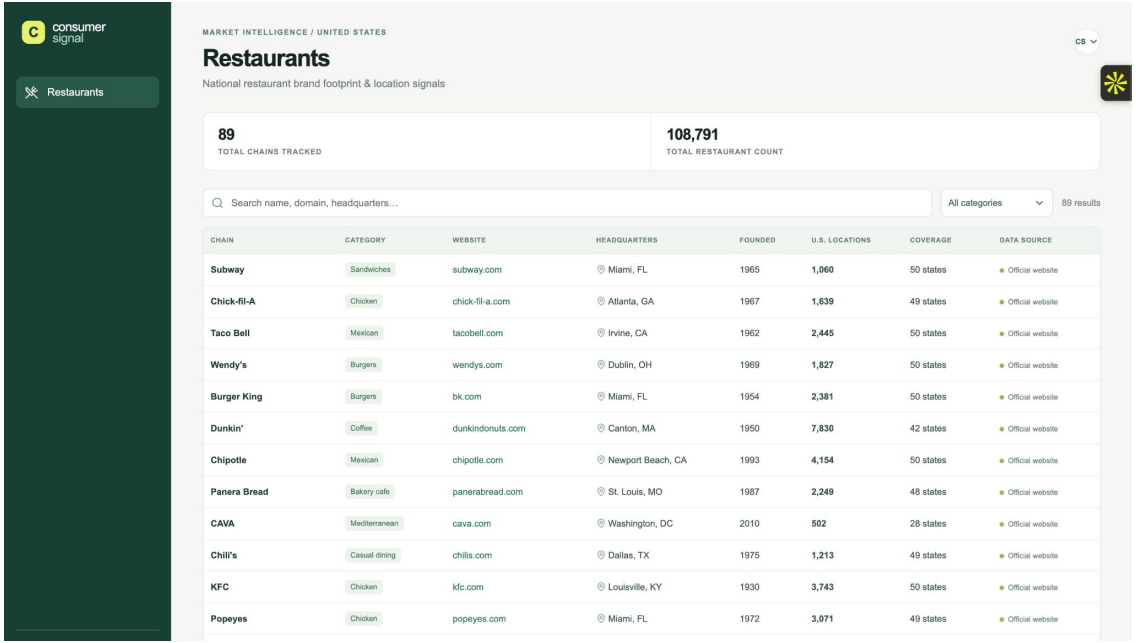
https://podcasts-xi.vercel.app/overview

Sample Project (2)



https://crunchmini.vercel.app/vc-firms

Sample Project (3)



https://consumer-signal-three.vercel.app/