Vigilo — Senior Engineer Feedback Analysis

Context

Notes from a conversation with a senior AI engineer, analysed through a VC and strategic product lens. Each note is broken down into what it means for Vigilo's architecture, defensibility, and roadmap.

---

1. "API Across System for Task Manager"

What he means: Don't build a monolithic monitoring agent. Build Vigilo as an API-first platform where components talk to each other through well-defined interfaces — like a task manager that orchestrates monitoring, alerting, and action.

Why it matters: This is a fundamental architecture decision. If you build it as a single Python script that watches processes, you'll hit a ceiling fast. If you build it as a set of APIs — a metrics API, an alerting API, an action API — you can plug any frontend (phone app, CLI, web dashboard) into the same backend.

Action for Vigilo:

- Design a REST/WebSocket API layer between the agent and the relay server
- The agent exposes a local API (e.g., `localhost:8420/metrics`, `/tasks`, `/status`)
- The relay server aggregates across machines and forwards to the phone app
- This means anyone could build a VS Code extension, CLI tool, or web dashboard on top of Vigilo's API later

---

2. "Start with Email to Access Data"

What he means: Use email-based authentication as your first auth layer. When someone installs Vigilo, they register with an email. That email becomes the key that links their agent → relay → phone app.

Why it matters: It's the simplest auth model that actually works. No OAuth complexity, no API key management for v1. Email gives you a user identity, which gives you data persistence, which gives you run history — one of your core features.

Action for Vigilo:

- On first run, prompt for email
- Send a verification token (Firebase Auth or simple magic link)
- All subsequent data is keyed to that email
- The phone app logs in with the same email and sees all connected machines

---

3. "Consider Using Packages and Integrating with First Python Packages"

What he means: Don't reinvent monitoring — `psutil` is just the start. There's a rich ecosystem of Python packages for system monitoring, process management, and ML-specific hooks. Lean into that.

Why it matters: Python is the lingua franca of your target users. If Vigilo's agent is a `pip install vigilo` away, with clean Python APIs that integrate with existing workflows, adoption friction drops to near zero.

Key packages to integrate with:

- `psutil` — CPU, RAM, disk, network (you already have this)
- `gputil` / `pynvml` — NVIDIA GPU monitoring
- `torch.cuda` — PyTorch-specific CUDA memory tracking
- `tqdm` — progress bar interception (millions of ML scripts use this)
- `wandb` / `mlflow` — integration hooks for experiment trackers
- `subprocess` — wrapping long-running processes for stdout capture

Action for Vigilo:

- Package the agent as `pip install vigilo`
- Provide a decorator/context manager: `with vigilo.watch(): train_model()`
- Auto-detect if PyTorch/TensorFlow is running and hook into GPU metrics

---

4. "Consider .NET and Check with JavaScript So It Can Run Across Systems"

What he means: Think about cross-platform from day one. The relay server should be in Node.js (JavaScript) so it runs anywhere. Consider that some enterprise environments use .NET. Don't lock yourself into a single stack.

Why it matters: Your current architecture has a Python agent + Node relay. That's actually ideal. Python agent runs where the ML code runs. Node relay is cross-platform and easy to deploy. The phone app is React Native (also JavaScript). This gives you a unified JS stack for everything except the agent itself.

Action for Vigilo:

- Python agent → stays Python (matches the user's environment)
- Relay server → Node.js / Express or Fastify (cross-platform, easy WebSocket support)
- Phone app → React Native (already planned)
- Future: Consider a lightweight .NET agent for enterprise Windows environments (Month 5-6 stretch goal, not MVP)

---

5. "On Mac — Consider Progress, Explore macOS miniOS, Health on Chip Trackers"

What he means: macOS has deeply integrated system health APIs that go beyond what `psutil` can see. Apple Silicon chips have dedicated health monitoring at the hardware level. Explore `powermetrics`, `macmon`, and the IOKit framework.

Why it matters: A huge chunk of your ML practitioner audience is on Mac. If Vigilo can show Apple Silicon GPU utilisation, thermal throttling, and neural engine usage — things that no other free tool surfaces to a phone — that's a genuine differentiator.

Key Mac-specific tools:

- `macmon` — real-time Apple Silicon monitoring (CPU, GPU, ANE, memory, power)
- `powermetrics` — Apple's own command-line system profiler (requires sudo)
- `IOKit` — low-level framework for hardware health (temperature, fan speed, battery)
- `sysctl` — kernel-level stats
- Activity Monitor data (accessible via private APIs)

Action for Vigilo:

- Month 1-2: Use `psutil` (works on Mac for CPU/RAM)
- Month 3-4: Add `macmon` integration for Apple Silicon GPU/ANE metrics
- Month 5-6: Explore `powermetrics` for thermal data, investigate IOKit bindings

---

6. "Agents to Interact with System Level — Make Decisions Based on Resource Usage and Limits"

What he means: This is the biggest idea. Don't just *monitor* — let Vigilo *act*. An AI agent that can observe system state and make decisions: pause a process before OOM, checkpoint a model, reduce batch size, kill a zombie process.

Why this changes everything: This is the difference between a dashboard and an autonomous system. A dashboard tells you "RAM is at 92%." An agent says "RAM is at 92%, your model will OOM in ~12 minutes. I've automatically triggered a checkpoint and will reduce batch size from 32 to 16 after this epoch."

This is exactly what we discussed yesterday — moving from passive monitoring to active intervention. The progression looks like:

Level	Capability	Example
L0	Observe	"CPU at 85%"
L1	Alert	"Warning: RAM at 92%, OOM in ~12 min"
L2	Recommend	"Suggest: checkpoint now, reduce batch size"
L3	Act (with consent)	"Shall I pause training and save checkpoint?"
L4	Act (autonomous)	"Auto-checkpointed at epoch 7. Reduced batch size. Training continues."

Action for Vigilo:

- Month 1-2 (MVP): L0 + L1 — observe and alert
- Month 3-4 (Intelligence): L2 — rule-based recommendations
- Month 5-6 (Polish): L3 — actionable phone buttons ("Pause", "Checkpoint", "Kill")
- Future: L4 — autonomous agent with user-defined policies

---

7. OpenClaw — What It Is and How It Applies

What it is: OpenClaw (formerly ClawdBot/MoltBot) is an open-source AI agent framework created by Peter Steinberger. It runs locally on your machine and connects to LLMs (Claude, GPT, DeepSeek) through messaging platforms (Slack, Discord, Telegram, WhatsApp).

Why your engineer mentioned it: OpenClaw has a `process-watch` skill that monitors system processes — CPU, memory, disk I/O, network, open files, port bindings, and process trees. It can watch for threshold violations and even kill runaway processes. This is *very close* to what Vigilo does.

How to integrate/apply:

- OpenClaw proves the concept that AI agents can manage system resources effectively
- Vigilo could expose itself as an OpenClaw skill — users who already run OpenClaw could add Vigilo as a monitoring plugin
- The `exec` and `process` tools in OpenClaw show the pattern: background a monitoring task, poll for status, take action based on thresholds
- OpenClaw's architecture (local agent → messaging interface) mirrors Vigilo's (local agent → phone app)

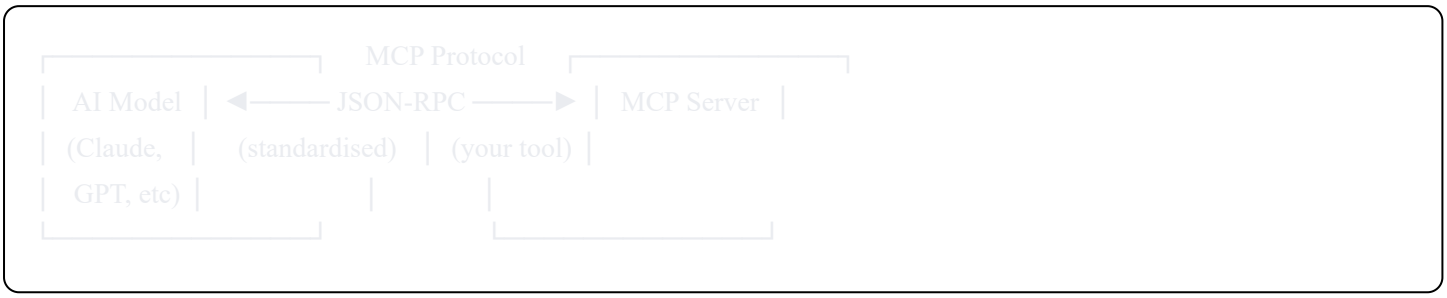
Key takeaway: OpenClaw validates the market. People want AI agents that interact with their system. Vigilo's advantage is that it's *purpose-built* for code execution monitoring with ML-specific intelligence, while OpenClaw is a general-purpose assistant.

---

8. MCP (Model Context Protocol) — What It Is and How It Applies

What it is: MCP is an open standard created by Anthropic (November 2024) for connecting AI systems to external data sources and tools. Think of it as "USB for AI" — a universal protocol that lets any AI model talk to any tool through a standardised interface.

How MCP works:



MCP defines three primitives:

1. **Tools** — functions the AI can call (model-controlled)
2. **Resources** — data the AI can read (app-controlled)
3. **Prompts** — templates the user can invoke (user-controlled)

How to apply MCP to Vigilo:

Build Vigilo as an MCP server. This means any AI assistant (Claude, GPT, Cursor, Windsurf) could directly query your machine's status:

```
User → Claude: "How's my training run going?"
Claude → Vigilo MCP Server: tool_call("get_training_status")
Vigilo MCP Server → Claude: { "cpu": 87, "ram": 92, "gpu": 78, "eta": "~2h 14m", "risk": "OOM in ~18 min" }
Claude → User: "Your training is 78% through but RAM is at 92%. I'd recommend checkpointing now."
```

Vigilo as an MCP server would expose:

- `get_system_metrics` → CPU, RAM, GPU, disk
- `get_running_processes` → list of active processes with resource usage
- `get_training_status` → ML-specific: epoch, loss, ETA, progress
- `predict_oom` → time-to-OOM prediction
- `checkpoint_model` → trigger a safe checkpoint (L3 action)
- `adjust_batch_size` → modify training parameters (L3/L4 action)

Why this is strategically brilliant: If Vigilo is an MCP server, it becomes part of the AI tooling ecosystem overnight. Any Claude Code user, any Cursor user, any OpenClaw user can connect to Vigilo. You don't need to build every interface yourself — the ecosystem builds on top of you.

Implementation path:

python

```python
# vigilo_mcp_server.py (simplified)
from mcp import Server, Tool

server = Server("vigilo")

@server.tool("get_system_metrics")
async def get_metrics():
    """Get current CPU, RAM, GPU, and disk usage."""
    return {
        "cpu_percent": psutil.cpu_percent(),
        "ram_percent": psutil.virtual_memory().percent,
        "gpu_percent": get_gpu_usage(),
        "disk_percent": psutil.disk_usage('/').percent
    }

@server.tool("predict_oom")
async def predict_oom():
    """Predict time until out-of-memory based on current growth rate."""
    return oom_predictor.estimate()

@server.tool("get_running_tasks")
async def get_tasks():
    """List all monitored long-running processes with status."""
    return task_manager.list_active()
```

Revised Architecture (Incorporating All Feedback)

Updated 6-Month Roadmap

Month	Focus	Key Deliverables
1	Foundation	Python agent (`pip install`), psutil metrics, email auth, WebSocket relay, basic React Native app
2	Connectivity	Push notifications (FCM/APNs), live metric streaming to phone, basic OOM detection
3	Intelligence	OOM prediction engine, stdout/tqdm progress parsing, rule-based recommendations
4	Platform	MCP server implementation, macmon integration (Mac), API documentation
5	Agency	Action buttons on phone (Pause/Checkpoint/Kill), policy-based auto-actions, run history + comparison
6	Polish + Deploy	TestFlight beta, App Store submission, OpenClaw skill publication, documentation site

---

Strategic Impact

Your senior engineer's feedback fundamentally repositions Vigilo:

Before: A monitoring tool that sends phone alerts. **After:** An agentic platform that observes, predicts, recommends, and acts on system resources — accessible from any AI interface via MCP, any messaging platform via OpenClaw, and your own phone app.

That's not a side project. That's a platform play. And the beautiful part is you can still build it in 6 months part-time, because each layer builds cleanly on the last.

The MCP integration alone could drive organic adoption faster than any marketing — every time someone connects Vigilo to Claude Code or Cursor and asks "how's my training going?", it just works. That's the kind of distribution that compounds.