Vibe Coding & Swarm Prototyping by Michaël Bettan

01. What is Vibe Coding? (#section-1)

Vibe Coding is an AI-assisted development approach where the developer describes what they want in natural language, and an AI coding assistant generates the implementation. Coined by **Andrej Karpathy** (co-founder of OpenAI) in early 2025, the term captures a paradigm shift: instead of writing code character by character, you *converse* with an AI to produce working software.

The developer's role shifts from **typing code** to **describing intent**, **reviewing output**, and **iterating through dialogue**. You "vibe" with the AI — expressing what you want, accepting suggestions, nudging direction, and occasionally correcting course.

Core Principles of Vibe Coding

- **Natural Language First:** You describe features, bugs, or architecture in plain English — the AI translates to code.
- **Conversational Iteration:** Development happens through a dialogue loop: `prompt → generate → review → refine`.
- **Intent Over Syntax:** Focus shifts from *"how to write it"* to *"what should it do"* and *"why"*.
- **AI as Pair Programmer:** The AI acts as an always-available coding partner with broad cross-stack recall.
- **Human as Architect:** You make design decisions, enforce boundaries, and own production quality; the AI handles implementation details.

> KEY INSIGHT
> Vibe coding doesn't mean "no coding knowledge needed." The best vibe coders are experienced engineers who leverage AI to amplify their expertise — they know **what to ask for**, how to **evaluate the output**, and when to **intervene manually**.

02. The Vibe Coding Spectrum (#section-2)

Vibe coding isn't binary — it exists on a spectrum from fully AI-driven to human-led with AI assistance. Understanding where you sit on this spectrum determines your workflow, sandbox isolation, and quality gates.

Mode	Operating Model	Appropriate Use Cases	Risk Level
Full Autopilot	AI generates everything; developer barely reads the code. Maximum speed, maximum risk.	Throwaway prototypes, hackathon demos, disposable personal scripts (inside a sandbox)	**Very High**
Collaborative Mode	Developer and AI co-create; human reviews every block and steers architecture.	Feature development, internal tools, startup MVPs	**Moderate**
Human-Led + AI Assist	Developer owns core logic & specs; AI handles boilerplate, tests, refactors, and docs.	Production systems, security-critical code, regulated industries	**Low**

> KARPATHY'S OWN WARNING
> Even Andrej Karpathy cautioned that vibe coding *"is not for production systems"* in its purest form. The more you move toward autopilot without specifications, sandboxes, and verification gates, the faster you accumulate **technical debt** and **hidden security vulnerabilities**.

The 5-Step Vibe Coding Loop

1. **Describe:** Write a natural-language prompt describing the feature, constraint, or bug fix.
2. **Generate:** AI produces code, architecture, or a working component.
3. **Review:** Developer reads, tests, and evaluates the generated diff.
4. **Refine:** Provide follow-up prompts to fix edge cases, add features, or refactor.
5. **Ship:** Integrate into the repository, run automated CI/security gates, document, and deploy.

03. What is Swarm Prototyping? (#section-3)

Swarm Prototyping takes vibe coding to the next level by deploying **multiple AI agents simultaneously** to build different variants, components, or approaches in parallel. Instead of one human conversing with one AI, you orchestrate a *swarm* of agents — each tackling a different aspect of the problem in an isolated workspace — and then select, merge, or compose the best results.

Think of it as **A/B testing during development**: spawn 5 agents to build 5 different implementations of the same feature, evaluate them all against a test harness, and ship the winner — or assign each agent a distinct microservice and build the backend concurrently.

Key Characteristics of Swarm Prototyping

- **Parallel Execution:** Multiple agents work concurrently on different tasks or implementation variants.
- **Diverge-then-Converge:** Generate multiple candidate options in parallel, evaluate objectively, and select the winner.
- **Specialization:** Each agent is scoped to a specific domain (frontend, backend, test harness, security audit).
- **Orchestration Layer:** A coordinating agent or human decomposes tasks, manages isolated branches/worktrees, and merges outputs.
- **Disposable Outputs:** Most candidate code is intentionally throwaway — the value lies in discovering the best architecture rapidly.
- **Scale-by-Agents:** Increase exploration throughput by adding parallel agents rather than serial human hours.

When to Use vs. When NOT to Use Swarm Prototyping

When to Use Swarm Prototyping	When NOT to Use Swarm Prototyping
Exploring multiple architectural approaches	Tightly coupled systems requiring sequential design

When to Use Swarm Prototyping	When NOT to Use Swarm Prototyping
Building proof-of-concepts under time pressure	Regulated / compliance-critical production software without review
Hackathons and innovation sprints	When merge/integration costs exceed generation savings
Generating UI/UX design variants	Projects requiring deep proprietary context the AI lacks
Benchmarking different algorithm implementations	When you lack an automated test/eval harness to score outputs
Building decoupled microservices in parallel	Production systems without human code review

04. Multi-Agent Architecture (`#section-4`)

Swarm prototyping relies on **multi-agent systems (MAS)** — architectures where independent AI agents collaborate, compete, or work in parallel:

```
[ Human Orchestrator ] ⟶ [ Coordinator Agent ] ⟶ [ Task Decomposition ] ⟶ [ Agent Swarm (N Workers) ] ⟶ [ Critic & Merger ]
```

Core Swarm Roles

- **Orchestrator Agent:** Central coordinator (human or AI) that decomposes tasks, provisions isolated worktrees/sandboxes, assigns work, and coordinates synthesis.
- **Worker Agents:** Specialized execution agents (e.g., UI builder, API implementer, pytest author, migration writer).
- **Critic / Evaluator Agent:** Reviews worker outputs in a clean context window, runs unit/security suites, scores quality, and ranks variants.
- **Merger Agent:** Integrates winning outputs from multiple branches into a cohesive codebase, resolving interface and schema conflicts.

Canonical Swarm Patterns

- **Fan-Out / Fan-In:** One large decomposable task → split across *N* parallel agents → merge results (map-reduce for code).
- **Tournament:** *N* agents compete independently on the exact same specification → deterministic test suite + critic picks the winning implementation.
- **Pipeline:** Agents execute sequentially across quality stages — `Agent 1 (Design/Spec) → Agent 2 (Implement) → Agent 3 (Test & Harden)`.
- **Specialization Grid:** Each agent owns a bounded domain (`UI` , `API` , `DB` , `Auth`) against a shared OpenAPI/schema contract.
- **Ensemble:** Multiple agents audit or solve the same critical problem (e.g., security review) and aggregate findings for high recall.

05. Vibe Coding vs. Swarm Prototyping (`#section-5`)

Dimension	Vibe Coding	Swarm Prototyping
Agents	Single AI assistant (1 human : 1 agent)	Multiple coordinated agents (1 human : *N* agents)
Workflow	Conversational, iterative, sequential	Parallel, orchestrated, diverge-then-converge
Context Ownership	Human maintains full session context	Context split across isolated subagent windows
Speed	Fast for focused single tasks	Faster for broad exploration & multi-service builds
Exploration	One path at a time	Many candidate paths simultaneously
Complexity & Setup	Low setup, immediate start	Higher setup (worktree isolation, eval harness, orchestration)
Token Cost	Low (single session)	Higher (~4×–15× token multiplier across *N* workers)
Integration	Naturally coherent in one branch	Requires explicit merge & contract verification step
Best For	Features, bug fixes, scripts, refactors	Architecture tournaments, PoCs, parallel microservices

> **COMPLEMENTARY, NOT COMPETING**
> High-performing teams combine both: **Swarm Prototyping** during ambiguous design sprints to benchmark candidate architectures, and **Vibe Coding** (grounded by IDD specs) for focused day-to-day feature delivery.

06. Tools & Ecosystem (`#section-6`)

Vibe Coding & Agentic Development Surfaces

- **Google Antigravity:** Agent-first AI IDE, CLI (`agy`), and Manager surface powered by `gemini-4-argon` / `gemini-3.8-flash` , featuring Planning Mode (`/plan`), parallel subagents (`Workspace="branch"`), browser-in-the-loop verification, and terminal sandboxing.
- **Claude Code (Anthropic):** Agentic terminal & IDE coding assistant (`claude`) with tool use, sandboxed bash, hooks, Git worktree swarms (`claude -w`), and native **Vertex AI (`CLAUDE_CODE_USE_VERTEX=1`)** enterprise integration.
- **Gemini Code Assist & Gemini CLI:** Google Cloud's enterprise IDE assistant and open-source terminal agent with VPC-SC governance, codebase indexing, and IP indemnification.
- **Cursor:** AI-native IDE with inline chat, tab completion, multi-file editing, and background cloud agents.
- **GitHub Copilot:** Inline completions and coding agent integrated across VS Code, JetBrains, Neovim, and GitHub PRs.
- **Windsurf:** AI IDE with multi-step "Cascade" agentic flows.
- **Aider:** Open-source CLI pair programmer built specifically for git-tracked repositories.

- **Replit Agent:** Full-stack cloud agent that scaffolds and deploys web apps directly from natural-language prompts.

Swarm & Multi-Agent Frameworks

- **Google Agent Development Kit (ADK):** Code-first Python/TS/Go/Java framework with `ParallelAgent`, `SequentialAgent`, `LoopAgent`, and `LlmAgent` for deterministic + dynamic multi-agent swarms.
- **Claude Agent SDK:** Programmatic TypeScript/Python SDK for orchestrating isolated subagents (`Task`), custom MCP tools, and verification hooks.
- **OpenAI Swarm / Agents SDK:** Lightweight primitives for multi-agent handoffs and routines.
- **CrewAI:** Role-based multi-agent orchestration with structured task pipelines.
- **AutoGen (Microsoft):** Conversational multi-agent framework with customizable roles.
- **LangGraph:** Stateful graph-based agent orchestration with cyclic control, checkpoints, and HITL interrupts.
- **MetaGPT:** Multi-agent framework simulating software-company roles (PM, architect, engineer, QA).
- **Vertex AI Agent Builder / Google Agent Platform:** Managed enterprise control plane (`Agent Runtime`, `Agent Registry`, `Sessions`, `Gen AI Evaluation`) for deploying and governing agent fleets.
- **Amazon Bedrock Agents:** AWS managed multi-step agent orchestration service.

```python
# Swarm Prototyping — Tournament Pattern (Parallel Worktrees + Objective Grader)
from swarm import Orchestrator, Agent, Evaluator

task = "Build a REST API for user authentication with short-lived JWTs"

# Spawn 3 worker agents in isolated Git worktrees (Workspace="branch")
agents = [
    Agent(name="flask-agent",   style="Flask + SQLAlchemy",  workspace="branch"),
    Agent(name="fastapi-agent", style="FastAPI + Pydantic",  workspace="branch"),
    Agent(name="express-agent", style="Express.js + Zod",    workspace="branch"),
]

# Fan out concurrently without shared .git/index collisions
results = Orchestrator.fan_out(task, agents)

# Deterministic test suite + independent Critic scores each variant
best = Evaluator.rank(results, criteria=["security", "code_quality", "test_coverage"])
print(f"Winner: {best.name} — Score: {best.score}")
```

07. Risks, Limitations & Best Practices (`#section-7`)

Risks & Anti-Patterns

- **Cargo Cult Coding:** Accepting AI output without understanding it — code passes a happy-path demo, but nobody on the team can debug or extend it.
- **Hidden Technical Debt:** AI generates duplicated, "good enough" imperative glue that silently erodes modularity.
- **Security Blind Spots:** Unchecked AI code frequently introduces SQL injection, SSRF, XSS, insecure deserialization, or hardcoded credentials.
- **Hallucinated APIs & Slopsquatting:** AI invents non-existent SDK methods or imports hallucinated package names that attackers register on PyPI/npm (**slopsquatting**).
- **Context Window Overflow & Rot:** Dumping massive repositories into one chat degrades attention and causes contradictory edits across files.
- **Skill Atrophy:** Uncritical reliance on Full Autopilot erodes engineers' mental models of system failure modes.

Core Best Practices

- **Always Review:** Read every line of AI-generated diffs before merging — treat AI output as an untrusted draft PR.
- **Test First:** Generate or write executable unit/integration tests alongside specifications before accepting implementation code.
- **Constrain Context:** Scope prompts with explicit architectural boundaries, pinned SDK versions, and file paths.
- **Version Everything:** Commit small, atomic checkpoints so bad agent trajectories can be reverted cleanly.
- **Automated Security Verification:** Run SAST, dependency validation, and PoC-verified remediation (**CodeMender**) on all generated code.
- **Document Intent:** Commit the specification or ADR capturing *why* the design was chosen, not just the resulting code.

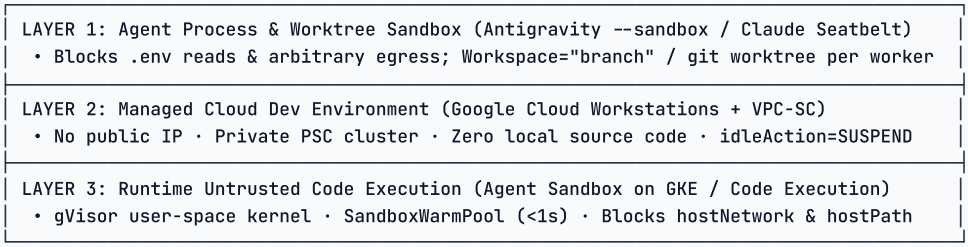
> **THE "IT WORKS" TRAP**
> The most dangerous outcome of vibe coding is code that **appears to work perfectly** in a local demo while hiding race conditions, multi-currency rounding bugs, or exploitable injection flaws. Always ask: *"Does it work, or does it only seem to work?"*

Production Readiness Checklist

- ✅ Every line of AI-generated code has been read and understood by a human reviewer
- ✅ Unit and integration tests cover critical paths and edge cases
- ✅ Security scan (SAST + secret detection) passes with zero injection vectors or hardcoded credentials
- ✅ Dependencies verified — all imported packages and API methods exist at approved, vulnerability-free versions
- ✅ Architecture documented so engineers can maintain and debug the service independently
- ✅ Performance load-tested under expected `p95` / `p99` concurrency (AI optimizes for plausibility, not tail latency)
- ✅ Licensing & IP provenance verified against copyleft policy

08. Secure Coding Sandboxes: Antigravity, Cloud Workstations & GKE (`#section-8`)

Giving an autonomous coding agent—or a 5-agent swarm—unrestricted shell access on a developer laptop holding production `gcloud` tokens, SSH keys, and `.env` secrets creates catastrophic blast radius: a single prompt injection inside a README or MCP response can exfiltrate credentials, wipe host directories, or corrupt shared Git state. Enterprise agentic development (PAA Section 2.1 / CCAR-F Domain 3) enforces **defense-in-depth sandboxing** across local CLI/IDE terminals, cloud dev environments, and runtime code execution clusters.



Enterprise Sandbox Comparison Matrix

Sandbox Surface	Primary Use Case	Isolation Boundary & Network Controls	Key Operational / Exam Rules
Antigravity Terminal Sandbox & Branch Workspaces	Interactive IDE / CLI (`agy`) vibe coding & parallel subagent swarms	OS-level process sandbox (`--sandbox` / `enableTerminalSandbox`); blocks `.env` secret reads and outbound network by default (`read_url` domain allowlist); **`Deny > Ask > Allow`** rule precedence.	**Off by default in App UI presets** (even `"Full machine"` —must toggle `Enable Terminal Sandbox` explicitly); **auto-enabled in CLI `auto` mode**. Spawn parallel swarm workers with **`Workspace="branch"`** (isolated Git worktrees) to prevent `.git/index.lock` collisions.
Claude Code Sandboxed Bash + Vertex AI	Terminal/IDE agentic coding (`claude`) with enterprise cloud governance	macOS Seatbelt / Linux `bubblewrap` filesystem write cage (cwd only) + proxy-enforced domain allowlist; parallel swarms use `claude --worktree` (`-w`).	Set **`CLAUDE_CODE_USE_VERTEX=1`** inside Cloud Workstations to authenticate via Google Cloud IAM (zero static Anthropic API keys), keeping prompts inside regional **VPC-SC** perimeters and **Cloud Audit Logs**.
Google Cloud Workstations	Standardized, zero-trust enterprise developer & coding-agent cloud IDEs	Dedicated Compute Engine VM inside customer VPC with **zero public IP** (`constraints/compute.vmExternalIpAccess`), **Private Cluster** via **Private Service Connect (PSC)**, and **VPC Service Controls** perimeter blocking data exfiltration.	Configure **`idleAction=SUSPEND`** (not `STOP` /delete) + **keep-alive scripts (`/google/scripts/keep_alive.sh`)** detecting active agent PIDs so long-running agent builds aren't killed mid-task while idle workstations pause vCPU/RAM billing with RAM state preserved!
Agent Sandbox on GKE *(Pod-Based)*	Executing untrusted AI-generated code, dynamic tool scripts, or isolated multi-tenant coding agents at scale	Dedicated Kubernetes Pod per sandbox (`Sandbox` / `SandboxClaim` / `SandboxTemplate`) isolated by **gVisor (`runsc`)** user-space kernel; **Validating Admission Policy** blocks `hostNetwork` , `hostPID` , and `hostPath` .	Attach a **`SandboxWarmPool`** CRD to pre-warm gVisor pods for **`< 1 second`** claim latency on GKE Autopilot or Standard; supports custom `PodSpec` sidecars and GPUs/TPUs.
Agent Substrate on GKE *(Actor-Based)* vs. **Managed Code Execution**	High-density stateful agents (*Substrate*) vs. zero-ops Python calculation sandboxes (*Code Execution*)	**Agent Substrate** multiplexes stateful agents on shared warm GKE Standard pods with `<1s` memory/filesystem snapshot-restore; **Agent Platform Code Execution** runs zero-egress micro-containers (`<1s` start, 100 MB I/O, up to 14-day TTL).	Do **not** use actor-based Agent Substrate when untrusted multi-tenant code requires strict per-tenant kernel/Pod isolation—use **Agent Sandbox on GKE** with **gVisor** instead.

Critical Enterprise Sandbox Architecture Patterns

1. **Preventing Swarm Git Corruption (`Workspace="branch"` / `git worktree`):** When 4 swarm agents edit files and run `pytest` concurrently inside the *same* working tree, they overwrite each other's uncommitted edits and crash on `.git/index.lock` . Always isolate each worker in its own **Git worktree** (`Workspace="branch"` in Antigravity `invoke_subagent` , or `claude -w <name>` in Claude Code), which shares the underlying `.git` object store without duplicating disk space.
2. **Zero-Exfiltration Contractor & Regulated Coding (`Cloud Workstations + VPC-SC`):** For regulated finance/healthcare teams or external contractors, prohibit source checkouts on unmanaged laptops. Provision **Cloud Workstations** with private gateway ingress, disable browser clipboard/SSH file transfer via IAM & Chrome Enterprise Premium policies, route LLM calls (`Gemini Code Assist` , `Antigravity` , or `Claude Code on Vertex AI`) over **Private Google Access** inside a **VPC-SC** perimeter, and bind workstation service accounts with least-privilege IAM.

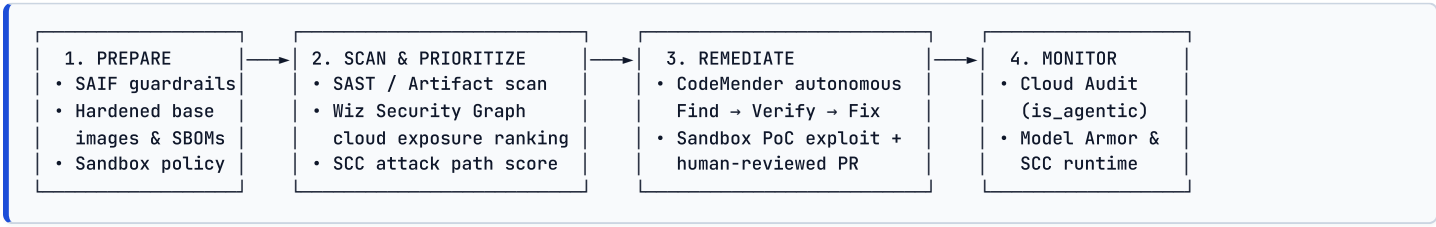
3. **Preserving Long-Running Agent Sessions Without 24/7 VM Burn:** Standard Cloud Workstation idle timeouts (`runningTimeout`, default 2 hours) terminate unattended VMs, killing background `/goal` or test-suite runs if no browser keystrokes occur. Combine a custom `/google/scripts/keep_alive.sh` hook (which returns busy while an `agy` / `claude` /build process is active) with **`idleAction=SUSPEND`** so truly idle workstations hibernate to persistent disk (halting compute charges) and resume with open terminals and memory intact.

09. Autonomous Vulnerability Remediation: CodeMender & Google AI Threat Defense (`#section-9`)

AI-assisted coding increases code output by 5×–10×, creating an acute security bottleneck: traditional SAST scanners bury AppSec engineers under thousands of **unverified false-positive alerts**, while naive "ask an LLM to patch this CVE" scripts hallucinate superficial fixes that break builds or leave the underlying flaw intact. **Google AI Threat Defense** and **CodeMender** (PAA Section 2.2) close this gap with **exploit-verified autonomous remediation**.

Google AI Threat Defense: The 4-Stage Security Lifecycle

Google AI Threat Defense organizes agentic and software security across four continuous stages:



1. **Prepare:** Establish Secure AI Framework (SAIF) policies, locked dependency registries, hardened container base images, VPC-SC perimeters, and sandbox templates before code is deployed.
2. **Scan and Prioritize:** Combine static/artifact scanning with **Security Command Center (SCC)** and the **Wiz Security Graph** to correlate code vulnerabilities against *real cloud runtime exposure* (internet ingress, over-privileged IAM service accounts, reachable sensitive data stores)—prioritizing exploitable attack paths over theoretical findings in dead code.
3. **Remediate:** Dispatch **CodeMender** to autonomously validate exploitability via sandboxed Proof-of-Concept (PoC) execution and synthesize root-cause, regression-tested patches for human PR approval.
4. **Monitor:** Continuously inspect live traffic and agent actions using **Model Armor** (prompt injection / PII / malicious URLs), **Dual-Attribution Cloud Audit Logs (`is_agentic=true`)**, and SCC runtime threat detection.

Inside CodeMender: Hybrid Architecture & The `Find → Verify → Fix` Loop

CodeMender (Public Preview, built by Google DeepMind) uses a **hybrid client-service architecture**:

- **Hosted Multi-Agent Reasoning Service:** Google-hosted Gemini reasoning agents (configurable via `--model` override) that hypothesize root causes, design exploits, synthesize patches, and run independent **LLM-as-judge** patch critiques.
- **Local CLI / Daemon Client (`cm`):** Runs directly inside the developer's workstation, CI runner, or private build container where the codebase and compiler toolchain live, equipped with a **process sandbox enabled by default** and explicit file scoping (`scan.extensions.include` in `config.yaml`; supports **C/C++, C#/.NET, Go, Java, JS/TS, Kotlin, Python, Ruby, Rust, and PHP**).

CodeMender Step	CLI Command	What Happens Under the Hood	Why Traditional Tools Fail Here
1. Find *(or Import)*	`cm find` **or** `cm report import --file ./findings.json`	Scans scoped source files for memory-safety, injection, auth-bypass, and logic flaws—**OR imports external SAST / Wiz Security Graph JSON reports** via `cm report import --file ./findings.json`.	Exam trap: CodeMender is **NOT limited to vulnerabilities it found itself**—it natively ingests third-party scanner findings to verify and patch them!
2. Verify *(PoC Exploit)*	`cm verify <FINDING_ID>`	Synthesizes, compiles, and executes a **concrete Proof-of-Concept (PoC) exploit / fuzzing harness inside an isolated sandbox (on by default)** to trigger the vulnerability deterministically.	SAST tools flag syntactic patterns with 40%–70% false-positive rates; `cm verify` proves real-world exploitability before bothering engineers.
3. Fix *(Root-Cause Patch)*	`cm fix <FINDING_ID> -y`	Traces dataflow/AST root causes, generates an idiomatic multi-file fix, **re-runs the PoC exploit** (asserting it now fails to exploit), runs the full **unit/regression suite**, passes an **LLM-as-judge** style & equivalence critique, and stages a **human-reviewed PR**.	Naive one-shot LLM patches add brittle input blacklists or break callers; CodeMender refuses to propose a patch unless the PoC is neutralized and tests pass.

```
# End-to-end CodeMender workflow in CI / Cloud Workstations
# 1. Import prioritized findings from external SAST / Wiz JSON report (or run `cm find`)
cm report import --file ./findings.json

# 2. Compile & execute sandboxed Proof-of-Concept (PoC) exploits to filter false positives (sandbox on by default)
cm verify FINDING_042

# 3. Synthesize root-cause patches, re-verify PoC neutralization + test suite, and stage PR
cm fix FINDING_042 -y --model gemini-4-argon
```

Architectural Guardrails for Autonomous Remediation

- **Mandatory Sandbox Execution for `cm verify` & `cm fix`:** Because `cm verify` compiles and executes live exploit payloads (buffer overflows, command injections, deserialization gadgets), always run verification inside CodeMender's **process sandbox** inside an isolated CI container, **Cloud Workstation**, or **Agent Sandbox on GKE (`gVisor`)**—never against live production databases or shared host credentials.
- **Human-in-the-Loop Merge Gate:** CodeMender automates root-cause localization, exploit verification, and patch validation, reducing mean-time-to-remediate (MTTR) from weeks to minutes—but **human maintainers retain final PR merge authority** on production branches, inspecting the attached root-cause explanation and regression PoC test.

10. Key Takeaways (`#section-10`)

Key Takeaways

- **Vibe Coding** is conversational AI-assisted development (`1 human : 1 agent`), coined by Andrej Karpathy in 2025, operating across a spectrum from **Full Autopilot** (throwaway prototypes) to **Human-Led + AI Assist** (production code).
- **Swarm Prototyping** orchestrates multiple specialized agents in parallel (`Fan-Out/Fan-In`, `Tournament`, `Pipeline`, `Specialization Grid`, `Ensemble`) to explore competing architectures rapidly; isolate workers in **Git worktrees (`Workspace="branch"` / `claude -w`)** to prevent `.git/index` collisions.
- The biggest danger of unverified AI code is the **"It Works" trap** (code that demos cleanly while hiding logic bugs, hallucinated APIs, or injection vulnerabilities). Always review, test, load-test, and scan AI-generated code.
- **Enterprise Sandboxing** layers **Antigravity Terminal Sandbox** (`--sandbox`, off by default in App presets; blocks `.env` & arbitrary egress; `Deny > Ask > Allow`), **Cloud Workstations** (private PSC cluster, zero public IP, VPC-SC, `idleAction=SUSPEND` + `/google/scripts/keep_alive.sh`), and **Agent Sandbox on GKE** (`gVisor` + `SandboxWarmPool` `<1s` startup).
- **CodeMender** operates in Stage 3 (**Remediate**) of **Google AI Threat Defense** (`Prepare → Scan & Prioritize → Remediate → Monitor`), executing **`cm find` / `cm report import` → `cm verify` (sandboxed PoC exploit eliminating false positives) → `cm fix` (PoC-neutralized, regression-tested, LLM-judge-critiqued PR for human review)**.
- Across both vibe coding and autonomous remediation, **the human shifts from typist to architect, reviewer, and final quality gate.**