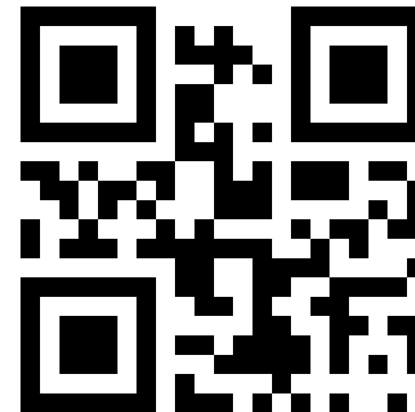
Stefano Maestri

maeste.it

Podcast: risorseartificiali.com

Newsletter: codiceartificiale.substack.com
artificialcode.substack.com

Backlog.md → github.com/MrLesk/Backlog.md
LINCE → github.com/RisorseArtificiali/lince



maeste.it

From Vibe to Agentic

The Coding You Didn't Know You Wanted

The silent revolution in software development

What is Vibe Coding?

> *"You just see stuff, say 'run', and it works. You can feel the code." - Andrej Karpathy*

The Approach

- Write a prompt, get code back
- Accept suggestions on instinct
- Iterate by describing what's wrong
- "It compiles? Ship it."
- Trust the AI output without deep review

Where It Works

- Prototyping, MVPs, hackathons
- Learning new languages/frameworks
- One-off scripts and automation
- **10x faster** for throwaway code
- Perfect when **correctness is not critical**

Where Vibe Coding Breaks

Security

AI-generated code often has vulnerabilities. SQL injection, XSS, hardcoded secrets. The "vibe" doesn't catch CVEs.

Maintainability

Code that "vibes" right today becomes legacy debt tomorrow. No architecture, no patterns, no consistency.

Correctness

"It works on my machine" scaled to "It works in the chat window". Edge cases, race conditions, data integrity.

Reliability

~50% agent task success rate without structure. Half the time the agent produces something broken or off-spec.

The core issue: **poor task specification**, not poor model capability.

The Evolution

Vibe Coding

"Make me a website"

AI generates code.
You hope it works.

~50% success rate

No constraints

Prompt Engineering

"Create a React app with TypeScript, use Next.js App Router, add error boundaries"

~70% success rate

Prompt constraints

Agentic Engineering

"Here are the specs, the harness, the tests. The agent operates within these guardrails."

~95% success rate

System constraints

The leap isn't in the model. It's in **the system surrounding the model**.

From Prompt to Harness Engineering

Prompt Engineering

- Crafting the perfect prompt
- Context stuffing
- One interaction at a time
- The human is the orchestrator
- Knowledge lives in your head

Harness Engineering

- Designing the operating environment
- Project rules (`CLAUDE.md`)
- Spec-driven task breakdown
- Automated validation & feedback loops
- Knowledge lives in the system

A good prompt helps for one turn. A good harness helps **permanently**.

The Missing Piece: Spec-Driven Development

The problem with "just implement feature X"

The agent doesn't know **what "done" looks like**. No acceptance criteria, no scope boundaries, no testable outcomes. It fills the gaps with assumptions.

Without specs

```
You: "Add user search"
Agent: builds something...
  - Wrong search algorithm?
  - Missing pagination?
  - No error handling?
  - Touches unrelated files?

WHO KNOWS. No spec = no way
to validate the output.
```

With specs

```
Task: "Add user search"
Acceptance Criteria:
  ✓ Full-text search on name/email
  ✓ Results paginated (20/page)
  ✓ Returns 404 on empty results
  ✓ Response time < 200ms
  ✓ Unit tests for all paths
Scope: only src/users/
```

Backlog.md: Specs for Agents

Backlog.md is an open-source, markdown-native task manager designed for human-AI collaboration. Tasks live as `.md` files in your repo. No SaaS, no external tools, 100% Git-native.

The 3-Phase Loop

1. Task Creation (Spec-Driven)

- Break work into atomic tasks
- Each task has acceptance criteria
- Written as "work order for a stranger"

2. Task Execution (Plan-First)

- Agent drafts plan before coding
- Human approves plan
- Only then: implement

3. Task Finalization (DoD)

- Verify all acceptance criteria
- Write PR-style summary
- Propose (don't create) follow-ups

Why It Works

- **One task per session, one PR per task**
- The creating agent is NOT the executing agent
- All context is self-contained in the task
- Human-in-the-loop at critical checkpoints
- Git-native audit trail

Key Rule

> "Write every task as a work order for a stranger who knows nothing about your conversation."

This forces completeness and eliminates assumptions.

Backlog.md in Action

```
# Task: Add Core Search Functionality

## Status: In Progress

## Description
Implement full-text search for users by name and email in the users API module.

## Acceptance Criteria
- [ ] GET /users/search?q=<query> endpoint
- [ ] Full-text search on name and email fields
- [ ] Results paginated (20 per page, configurable)
- [ ] Returns empty array (not 404) for no results
- [ ] Search is case-insensitive
- [ ] Response time < 200ms for 10k users
- [ ] Unit tests cover: valid search, empty result, pagination, special characters

## Scope
- ONLY modify files in src/users/
- Do NOT change database schema

## Implementation Plan (approved ✓)
1. Add search repository method with SQL ILIKE
2. Create search endpoint in routes.py
3. Add pagination helper
4. Write test suite
```

The agent reads this, implements exactly what's specified, and checks off criteria as it goes.

But One Agent Is Not Enough

Real projects need **parallel work streams**:
backend API, frontend UI, tests, documentation, security review...

The orchestration problem

```
Agent 1: working on backend API          → needs monitoring
Agent 2: working on frontend components  → needs monitoring
Agent 3: writing integration tests       → needs monitoring
Agent 4: updating documentation          → needs monitoring

Developer: ALT-TABbing between 4 terminals, losing sanity
```

Agentic engineering at scale requires **multi-agent orchestration**. You need a command center, not a pile of terminal tabs.

LINCE: The Agent Command Center

LINCE (Linux Intelligent Native Coding Environment) is a TUI dashboard for managing multiple Claude Code agents simultaneously. A Zellij WASM plugin built in Rust.

Core Features

- **Multiple agents** in one terminal
- Real-time status: Running, INPUT, Idle
- Token usage tracking per agent
- Active tool display (what is each agent doing?)
- Subagent count tracking
- Focus/hide agent panes with one key
- Session persistence (save & restore)

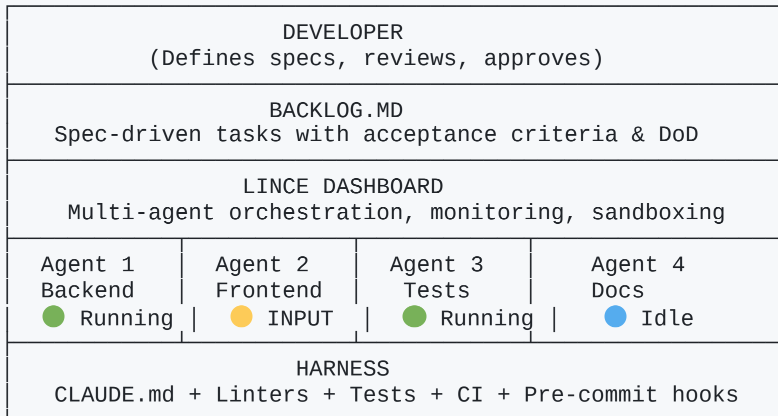
The Safety Layer

- Every agent runs in a **bubblewrap sandbox**
- Full write access to project directory
- **Zero access** to SSH keys, cloud credentials, home directory
- Enables `--dangerously-skip-permissions` safely
- Full autonomy within strict boundaries

Bonus

- **Voice control** via local Whisper
- Speak commands, routed to focused agent

The Full Stack: Backlog.md + LINCE



Specs (Backlog.md) + Orchestration (LINCE) + Guardrails (Harness) = **Agentic Engineering**

Demo: The Agentic Workflow

```
STEP 1 - SPEC (Backlog.md)
$ claude "Break down the user management feature into tasks"
→ Agent creates 4 tasks in backlog/ with acceptance criteria and scope

STEP 2 - REVIEW
Developer reviews tasks, adjusts scope, approves plans

STEP 3 - ORCHESTRATE (LINCE)
$ lince-dashboard
┌──────────────────────────────────────────────────┐
│ LINCE Dashboard           Agents: 3 | Tokens: 45k │
├──────────────────────────────────────────────────┤
│ 🟢 agent-backend  | Running | Edit src/users   | 12k │
│ 🟢 agent-tests    | Running | pytest           | 8k  │
│ 🟡 agent-frontend | INPUT   | waiting...       | 25k │
└──────────────────────────────────────────────────┘

STEP 4 - VALIDATE
Each agent runs tests, checks acceptance criteria, creates PR
Agent-frontend asks: "Should search be client-side or server-side?"
Developer answers, agent continues autonomously
```

Patterns vs Anti-Patterns

What Works

Spec-first development
Write the task spec before any code

One task, one agent, one PR
Atomic, verifiable units of work

Plan-then-execute
Agent drafts plan, human approves

Sandboxed autonomy
Full freedom within strict boundaries

Feedback loops
Tests + linting + CI as teacher

Scope containment
"Only modify files in src/auth/"

What Doesn't Work

The Mega-Prompt
Stuffing everything into one prompt

YOLO Mode
"Just do it, I trust the AI"

Micromanagement
Dictating every step to the agent

No Feedback Loop
Agent produces output, nobody checks

Scope-Free Agents
No boundaries = unpredictable changes

One-Shot Thinking
Expecting perfection on first try

The Developer's New Role

What You Stop Doing

- Writing boilerplate code
- Manual formatting and syntax
- Being the human compiler
- Alt-tabbing between terminals
- Holding context in your head

What You Start Doing

- Writing specs and acceptance criteria
- Designing harnesses and guardrails
- Reviewing agent plans and output
- Orchestrating parallel work streams
- Being the system architect

Before	After
"I can write React"	"I can write specs that agents implement correctly"
"I know Python syntax"	"I can design validation pipelines"
"I debug by reading code"	"I debug by improving guardrails"
"I manage my TODO list"	"I manage a fleet of sandboxed agents"

What This Means For You Today

Start Now

- Add `CLAUDE.md` to your projects
- Install **Backlog.md**: specs for every task
- Write acceptance criteria, not vague requests
- Set up pre-commit hooks

Level Up

- Try **LINCE** for multi-agent work
- Design feedback loops (tests as teachers)
- Practice the plan-then-execute loop
- Sandbox your agents

Think Ahead

- Learn system design, not just syntax
- Practice declarative thinking
- Embrace constraint-driven design
- Build agent-friendly codebases

backlog.md → github.com/MrLesk/Backlog.md | **lince** → github.com/RisorseArtificiali/lince

Key Takeaways

1. Vibe coding: ~50% success. Spec-driven agentic: ~95%.
2. The new skill is harness engineering, not prompt engineering
3. Backlog.md: write specs as work orders for strangers
4. LINCE: orchestrate multiple sandboxed agents in parallel
5. Constraints make agents more reliable, not less capable

The coding you didn't know you wanted is the coding where you **design the system** instead of writing the code.

Thank You

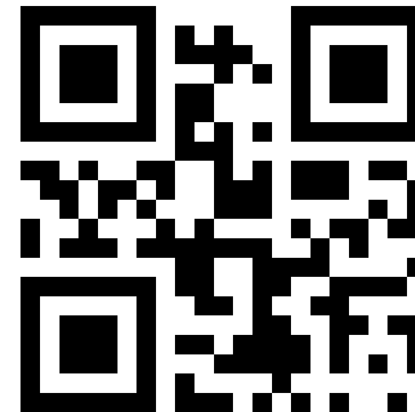
Stefano Maestri

maeste.it

Podcast: risorseartificiali.com

Newsletter: codiceartificiale.substack.com
artificialcode.substack.com

Backlog.md → github.com/MrLesk/Backlog.md
LINCE → github.com/RisorseArtificiali/lince



maeste.it

Questions?