Lecture 16: Vibe Programming and AI-Driven Software Development

1. Introduction: Vibe Programming Overview

- Guest lecture by Prof. Kerry Nachenberg, with a demonstration-based format and live coding.
- Focus: Vibe Programming—AI-assisted software development, current capabilities, and workflows of AI agents in software construction.

---

2. What is Vibe Programming?

2.1 Definition and Origin

- Term coined by Andrej Karpathy (AI expert).
- Karpathy's description:

  > "Fully give into the vibes, embrace exponentials, and forget that the code even exists."

- ChatGPT-3 definition:

  > "An informal AI-assisted approach where you prototype by chatting with a model, iterating code, tweaking prompts, and following intuition instead of detailed upfront specs."

- Aspirational: True "vibe" programming is not fully realized yet; debugging and requirements are still necessary.

Key Distinction:

- Vibe Programming is not about abandoning rigor or requirements—it's about leveraging AI to accelerate prototyping, iteration, and learning, while still applying critical thinking and validation.

2.2 "Embrace Exponentials"

- AI capabilities are improving at an exponential rate.
- Example: LLMs can now complete hour-long human tasks with much greater success than before; progress doubles roughly every 7 months.
- **Implication:** The tools and workflows you use today may be outdated in a year—adaptability is crucial.

---

3. The Spectrum of Vibe Programming

Phase	Description	Example Use Case
LLM Assistance	Use AI for small tasks (e.g., generating functions or boilerplate code).	"Write a Python function to parse CSV files."
Collaboration	Human and AI co-develop; AI handles sections, humans debug and integrate.	"Build a web app skeleton, I'll add business logic."
Full AI Generation	AI builds complete applications without human-written code (e.g., Pac-Man).	"Generate a playable Pac-Man clone from scratch."

- **Real-world adoption:** Y Combinator reports 20% of teams use AI for most of their product, with 95% of code written by AI in some startups.
- **Most effective today:** Collaboration, not full AI generation.

ASCII Diagram: Spectrum of Vibe Programming

```
[LLM Assistance] <--> [Collaboration] <--> [Full AI Generation]
    ^                    ^                        ^
    |                    |                        |
  Human-driven      Human+AI team           AI-driven, human reviews
```

---

4. Tools of Vibe Programming

4.1 Cursor (Modified VS Code)

- IDE based on VS Code, optimized for AI-assistance.
- Supports multi-file projects, shell access, testing integration.
- Eliminates manual file management (no more copy-pasting from ChatGPT).

4.2 Other Tools

- Claude Code

- Codex (OpenAI)
- Google's Gemini

4.3 Tool Selection Guidance

Tool	Best For	Limitations
ChatGPT	One-off scripts, simple files	No multi-file/project support
Cursor	Multi-file, iterative projects	Requires setup, learning curve
Claude/Gemini	Large codebases, code review, explanations	May lack IDE integration

Agentic vs. Non-Agentic Tools:

- **Agentic tools** (e.g., Cursor) can read/write files, run shell commands, and interact with your environment.
- **Non-agentic tools** (e.g., ChatGPT web) only generate text/code and require manual copy-paste.

---

5. Understanding Agentic AI

5.1 What is an Agent?

- A software system that couples:
  - An LLM (e.g., GPT-4, Claude)
  - A Tool/API Layer (e.g., filesystem, shell, calendar, process execution)
- Agents interpret LLM output (often JSON commands) and execute actions in the real world.

Example: Agentic vs. Non-Agentic Workflow

```
Non-Agentic:
[User] <-> [LLM] (text only)

Agentic:
[User] <-> [Agent (LLM + Tools)] <-> [Filesystem/Shell/External APIs]
```

5.2 Components in Agentic Framework

1. **Client (IDE):** Cursor (VS Code fork); interfaces with local files/shell.
2. **Backend Server:** Hosted by Cursor to enrich/route prompts and manage command schemas.
3. **LLM:** External model (OpenAI, Google, Anthropic) used to generate commands and logic.

5.3 Agent Commands (Examples)

Action	Command Format (JSON-like)
Google Search	{ "command": "search", "query": "what is pathfinding?" }
Read From File	{ "command": "read_file", "path": "src/main.ts" }
Write File	{ "command": "write_file", "path": "game.ts", "data": "..."}
Execute Shell Command	{ "command": "shell", "cmd": "npm start"}
Display Message	{ "command": "message", "text": "Task completed successfully!"}

- Agentic frameworks use a protocol (e.g., MCP) to standardize these commands.

---

6. Agentic Workflow Architecture

6.1 Flow of Execution

1. User issues a command (e.g., "Fix bug in foo").
2. Sent to backend server, which appends available command schemas.
3. Prompt sent to LLM.
4. LLM returns a command (e.g., read a file).
5. Backend interprets and executes the command via client.
6. Feedback (e.g., file contents) appended to prompt.
7. Repeat until desired output is attained.

Step-by-Step Example:

- User: "Add a login page."
- Agent reads `app.tsx`.
- Agent writes new `Login.tsx` file.
- Agent updates `routes.tsx`.
- Agent runs `npm install react-router-dom`.
- Agent notifies user: "Login page added."

6.2 Prompt Management

- Every interaction builds on prior ones; context accumulates.
- Accumulated context can confuse the model ("crud accretes").
- Solution: Re-initialize sessions regularly to avoid confusion.

Improved ASCII Diagram: Agentic Workflow (with context accumulation)

```
+--------+      +-------------+      +--------------+      +-----+
|  User  | ---> | Cursor IDE  | ---> | Cursor Server| ---> | LLM |
+--------+      +-------------+      +--------------+      +-----+
     ^                |                    |                |
     |                |<-------------------|<---------------|
     |                |   (context, files, |   (commands,   |
     |                |    feedback, etc.) |    responses)  |
     |                +------------------------------------+
     |                       (context accumulates)
     +-----------------------------------------------------+
```

---

7. Key Concepts Before Starting Vibe Programming

7.1 Coding Rules Configuration

- Specify constraints and style guidelines for the LLM.
- Examples:
  - Reuse/refactor existing code when possible.
  - Do not introduce new frameworks/technologies without confirmation.
  - Make minimal required changes.
- These rules are included in the prompt for every request.

7.2 Requirements Gathering

- Essential for successful Vibe Programming, especially for complex projects.

Table: Requirements Gathering Components

Component	Description
Functional Req.	What the software must do (UI, game physics, etc.)
User Stories	How users interact with the system
Framework/API Info	Preferred tech stack
UI Mockups	Visual layouts, wireframes, Figma diagrams
Business Rules	Backend logic and constraints
Testing Requirements	How the code is tested (unit tests, etc.)
Performance/Security	Runtime and safety expectations
Acceptance Criteria	Success conditions for tasks/project

Step-by-Step Example: Requirements to PRD

1. **Gather Requirements:**
   - Interview stakeholders, collect user stories, define business rules.
2. **Draft PRD:**
   - Write a Product Requirements Document with purpose, features, scope, stretch goals.
3. **Review and Refine:**
   - Share PRD with team, iterate based on feedback.
4. **Break Down Tasks:**

- Create a dependency-aware task list from PRD.

Example PRD Outline:

```
Product: 3-Player Pong Game
Purpose: Enable 3 users to play Pong simultaneously.
Features:
  - Three paddles, one for each player
  - Ball physics and collision
  - Score tracking
  - Stretch: Power-ups, AI opponent
Acceptance Criteria:
  - Game runs in browser
  - All paddles respond to input
  - Game ends when a player reaches 10 points
```

---

8. Demonstration Project: 3-Player Pong Game in TypeScript

8.1 Process Overview

1. **PRD Generation (Product Requirements Document):**
   - Formal structure: purpose, features, scope, stretch goals.
   - Can be generated by GPT using a task description.
2. **Dependency-Aware Task List:**
   - Created from PRD; includes ordered and split sub-tasks.
   - Example tasks: setup project, create main loop, paddle class, ball class, input handling, collision detection.
3. **Implementation using Cursor:**
   - Tasks executed in order via automated prompting.
   - Human triggers one task at a time; Cursor performs file creation, code generation, package installation, UI/gameplay updates.

8.2 Implementation Details

- Each task is executed in a fresh chat session to avoid context confusion.
- If a task goes "off the rails" (e.g., major bugs, wrong language/framework), revert to last checkpoint and try again.
- Validate each step by running the code and checking results.
- Use diagnostics/logging and model switching for debugging.

ASCII Diagram: Task Execution Loop

```
[Start Task] -> [Fresh Chat] -> [Model Executes Task] -> [Validate Output]
      ^                                                        |
      |                                                        v
[If Off Rails: Revert] <------------------------ [If Bug: Debug/Retry]
```

---

9. Debugging with Vibe Programming

9.1 Bugs and Failures

- "Off-rails": Non-functional or highly illogical generations.
- Partial bugs: Small issues amidst working components.

9.2 Debugging Procedure

1. Start a clean chat session.
2. Clearly describe the issue.
3. Ask for multiple root cause suggestions (not just one).
4. Add diagnostics (e.g., logging) as suggested.
5. Run the program and gather output/logs.
6. Request a targeted fix based on new information.
7. If the fix fails, revert to checkpoint and/or try a different model.

Table: Debugging Workflow

Step	Action	Purpose

Step	Action	Purpose
1	Start clean chat session	Avoid context confusion
2	Describe the issue	Give the model clear context
3	Ask for multiple root causes	Get a range of hypotheses
4	Add diagnostics/logging	Gather more information
5	Run and collect output	Provide evidence for next step
6	Request targeted fix	Focus on the actual problem
7	Revert/try different model if needed	Avoid compounding errors

Debugging Example:

- Issue: Game crashes when ball hits paddle edge.
- Steps:
    1. Start new chat, describe bug.
    2. Ask: "What are 3 possible causes?"
    3. Add logging to collision function.
    4. Run and collect logs.
    5. Ask for targeted fix based on logs.

9.3 Debugging Strategies

- Avoid "fix on top of fix"; revert instead.
- Use diagnostics to iteratively narrow down the issue.
- Try different LLMs (Claude, Gemini, etc.) for fresh perspectives.
- Use revert/restore for checkpointing.

---

10. Pro Tips for Effective Vibe Programming

Tip	Explanation
Use Correct Language and Platform	AI performs best in familiar environments (e.g., TypeScript for web, Python for ML).
Add Tests to Requirements	Guides AI to generate better, verifiable code.
Avoid Fixes on Fixes	Causes accumulation of errors; revert instead.
Paste API Docs	Enables LLM to better call external services reliably.
Ask for Refactoring	AI can improve and clean up code it has previously written.

- Choose the right language/platform for the task (e.g., TypeScript for games, Python for ML).
- Add testing requirements to ensure code is verifiable and self-debugging.
- Paste API documentation directly into the project for better integration.
- Ask for code refactoring to improve maintainability and reduce duplication.

---

11. Future of AI in Software Development

11.1 Job Market Perspective

- AI can't fully replace human intuition, system design, or novel reasoning (yet).
- Developer roles may shift from writing to orchestrating and reviewing code.
- Increasing importance of product thinking: defining requirements clearly.
- Collaboration, not handoff: AI is a tool, not a replacement.
- **Q&A Insight:** The future of software jobs is not "no code" or "all code"—it's a blend. Not using AI is a disadvantage, but over-reliance leads to poor code quality. Human review and understanding remain essential.
- **Domain expertise** is critical for security and correctness. Use AI for productivity, but always review and understand the output, especially for security-sensitive or complex systems.

11.2 Educational Use

- Ideal for learning/understanding concepts and debugging unfamiliar code.
- Over-dependence can hinder foundational coding skill development.
- Use AI to explain code and concepts, not just to generate code.
- **Advice:** Use AI to help understand, not just to code. For learning, generate code with AI, then ask for line-by-line explanations to build intuition.

- **Warning:** Studies show that using AI for code completion can hurt learning and intuition if overused. Use it to supplement, not replace, your own problem-solving.

Learning Strategies Table:

Strategy	Benefit	Risk if Overused
Generate code, then explain	Deepens understanding, builds intuition	May skip critical thinking
Use AI for debugging unfamiliar	Accelerates learning, exposes new patterns	May not learn root cause analysis
Ask for refactoring suggestions	Improves code quality, exposes best practices	May not learn to refactor solo
Rely only on completions	Fast, but shallow learning	Weakens problem-solving skills

---

12. Empirical Data on AI in Software Development (Dr. Eggert's Segment)

12.1 JetBrains 2024 Developer Survey

- 63% of developers save less than 4 hours/week using AI tools (10% productivity boost for a 40-hour week).
- Main benefits: less time searching, faster coding, easier repetitive tasks.

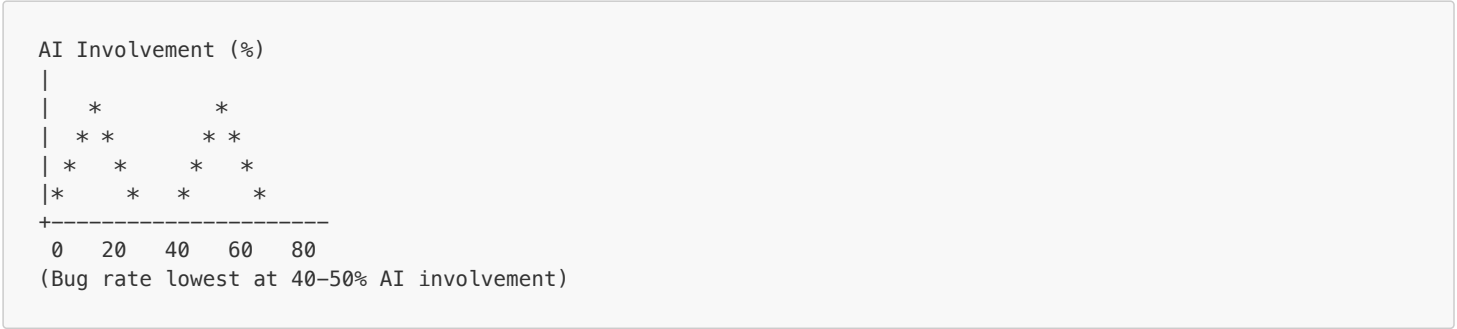
12.2 Meta / Satya Nadella Observations

- Microsoft: 20–30% of repo code is AI-generated; Meta aims for >50%.
- Meta's internal study: AI involvement from 10% to 80% of code; lowest bug rates at 40–50% AI involvement.
- AI excels at:
  - Boilerplate/framework code
  - Large-scale refactoring
  - Migration scaffolds
  - Test case generation
- AI struggles with:
  - Complex logic paths
  - Context-heavy features
  - Real systems thinking/architecture
  - Stateful or edge-case-heavy code

Table: AI Strengths and Weaknesses

Strengths	Weaknesses
Boilerplate/framework code	Complex logic paths
Large-scale refactoring	Context-heavy features
Migration scaffolds	Real systems thinking/architecture
Test case generation	Stateful/edge-case-heavy code

ASCII Diagram: AI Involvement vs. Bugs



12.3 Key Concerns

Concern	Explanation	Actionable Advice
Privacy & Data Leakage	Risk of sending proprietary code to external models.	Use private/incognito modes, avoid sensitive data.
IP Ownership	Who owns AI-generated code?	Check company policy, monitor legal developments.
Maintenance Complexity	Large, AI-written codebases may be harder to maintain.	Enforce code review, require documentation.

Concern	Explanation	Actionable Advice
Security Vulnerabilities	Risk of inserting exploitable code (e.g., SQL injection).	Use static analysis, review with a second model.
Data Poisoning Risk	Adversarial inputs training the AI to produce backdoors.	Use trusted sources, review generated code.

- **Additional issues:**
  - Leakage of private information (e.g., medical/student data, company secrets, student grades). Use private/incognito modes if available.
  - Unclear IP/copyright status of AI-generated code. Legal landscape is evolving; be cautious with proprietary or commercial projects.
  - Maintenance and operational costs may increase with AI-generated code due to bloat or lack of documentation.
  - Security: AI may generate insecure code, especially if trained on bad examples or adversarial code. Use a second model to review for vulnerabilities.
  - **Data poisoning:** Malicious actors could publish insecure code to "poison" AI training data, leading to vulnerabilities in generated code.

---

13. Summary

This lecture explored the evolving paradigm of Vibe Programming—a method of AI-assisted software development that leverages large language models and integrated agentic systems to generate and manage code. The concept, introduced by Andrej Karpathy, is built on a spectrum from simple LLM assistance to full AI-driven codebases. The lecture emphasized the importance of clear requirements, tooling like Cursor for agentic workflows, and formalized processes such as PRD and dependency-aware task lists.

It provided a deep dive into the architecture of agentic AI, demonstrating how JSON-based command exchanges between user, server, and model guide automated code generation and problem-solving within development environments. Debugging strategies, coding rules, and requirement gathering were also covered, showing how developers can maximize AI's utility while avoiding pitfalls like hallucinations, model confusion, and inappropriate code rewrites.

The session concluded with insights into the current and future job market implications, security risks, and empirical observations on AI effectiveness, suggesting that while AI tools are improving developer productivity, human oversight and understanding remain indispensable for building robust and secure systems.

Final Takeaways:

- Use AI as a collaborator, not a replacement.
- Always review, test, and understand AI-generated code, especially for security and correctness.
- Clear requirements and domain expertise are more important than ever.
- The legal, educational, and security landscape is rapidly evolving—stay informed and cautious.