Building Projects with LLM

Fundamentals of Machine Learning

Xufeng Zhang

Inria
xufeng.zhang@inria.fr

December 11, 2025

Agenda

1. Why AI for Programming Projects?
2. AI Tools Across the Project Lifecycle
3. Coding with AI Assistants
4. Debugging and Testing AI-Generated Code
5. Security, Governance, and Team Practices
6. Learning and Case Study
7. Wrap-Up

From Classical Coding to AI-Assisted Development

- Traditional workflow:
  - Human does **everything**: design, coding, tests, debugging.
- AI-assisted workflow:
  - Humans focus on **intent, architecture, and review**.
  - AI helps with **generation, exploration, and routine edits**.
- Goal: treat AI as a **collaborator**, not a magic box.
- Question for this talk: *How do we safely use AI to build real projects end-to-end?*

Why AI for Coding Now?

- Generative AI tools (e.g., code assistants, chat models) can:
  - Suggest code completions and entire functions.
  - Explain unfamiliar APIs and libraries.
  - Generate tests, documentation, and refactors.
- Studies on AI coding assistants show:
  - **Faster task completion** and higher perceived productivity.
  - Developers feel they can **focus on higher-level design** and more satisfying work.
- At the same time, AI-generated code can contain bugs and security issues.
- We must combine **productivity gains** with **engineering discipline**.

Evolution of Software Teams with AI

- Earlier: solo developer owns full project.
- Then: specialized teams (frontend, backend, infra, SRE).
- Now: developers work with **AI coding assistants** embedded in IDEs and terminals.
- Emerging: a single developer managing **multiple AI agents** that:
  - Refactor code, write tests, update docs.
  - Monitor production and propose fixes.
- New skill: **agent management** — telling AI *what* to do and validating *what it did*.

Landscape of AI Coding Tools

- **Inline code assistants**
  - IDE plugins, editor suggestions, autocomplete / in-fill.
- **Chat-based development**
  - Ask questions in natural language, paste code, get edits.
- **App builders and UI generators**
  - Prompt-to-app tools for full-stack or UI-heavy projects.
- **Agentic tools**
  - Multi-step "agents" that can run commands, modify files, run tests.
- **DevOps & SRE assistants**
  - Analyze logs, traces, alerts; propose mitigations in production.

AI Along the Project Lifecycle

Phase	AI can help with
Ideation	Brainstorming project ideas, feature sets
Design	Writing specs, proposing architectures, diagrams
Setup	Boilerplate, scaffolding, config files, CI templates
Coding	Implementing functions, refactors, API usage
Testing	Unit tests, integration tests, fuzz cases
Ops	Monitoring, log analysis, incident playbooks

Choosing the Right Tool for the Job

- Start from the **task**, not the tool.
  - "I want to design a REST API" vs. "I want to use X."
- Prefer:
  - Inline assistants for **small edits and completions**.
  - Chat for **design questions, explanations, larger diffs**.
  - App builders when you need a **quick prototype or UI**.
- For security- or safety-critical code:
  - Use AI as a **reviewer**, not the primary author.
  - Always apply additional static analysis and human review.

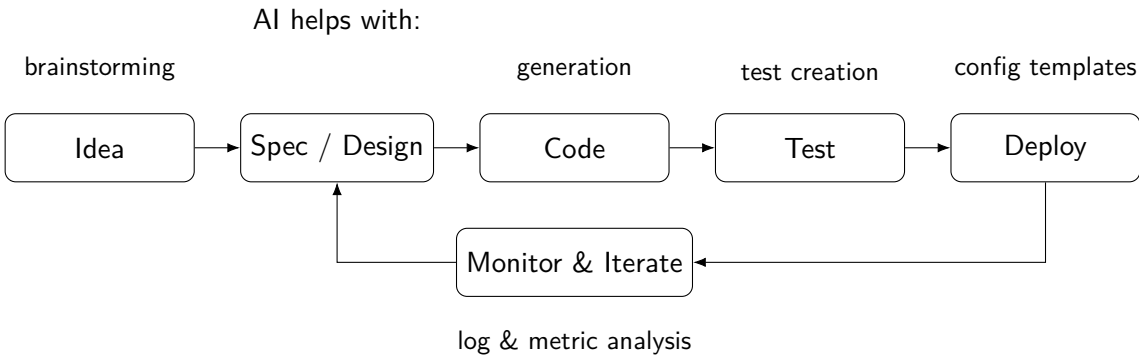
Using AI for Project Ideation

- Good prompts:
  - "Suggest 5 project ideas that combine computer vision and education."
  - "Propose beginner-friendly projects using Python and web APIs."
- Ask for:
  - **Scope** (MVP features vs. stretch goals).
  - **Complexity estimate** and prerequisite knowledge.
  - **Potential risks**: performance, data, security.
- Use AI-generated ideas as **starting points**, then:
  - Re-scope to fit your time and skills.
  - Align with course or business constraints.

Using AI for Requirements and Design

- Ask AI to help write:
  - User stories and acceptance criteria.
  - API contracts and data models.
  - Non-functional requirements (latency, scalability, privacy).
- Prompt patterns:
  - "Convert these notes into a concise design doc with sections for overview, data model, API, risks, and open questions."
  - "Propose 2 alternative architectures and compare trade-offs."
- Always review for:
  - Feasibility with your stack.
  - Overly complex suggestions for simple projects.

Diagram: AI-Assisted Project Workflow



Project Setup with AI

- Use AI to:
  - Generate minimal project scaffolding (folders, main files).
  - Propose dependency lists and basic configuration.
  - Suggest CI workflows (GitHub Actions, GitLab CI, etc.).
- Ask for explanations:
  - "Explain each file you created and how to run the project."
  - "Show me how to run tests and format the code."
- Immediately add:
  - `README`, `LICENSE`, and basic `.gitignore`.
  - Initial tests to serve as a safety net for future AI edits.

Prompt Patterns: Micro-Tasks

- Work in **small, clear steps**.
- Examples:
  - "Write a pure function in Python that validates an email address. Include 3 unit tests with edge cases."
  - "Refactor this function for readability without changing behavior."
- Provide context:
  - Language, framework, style guidelines.
  - Constraints: performance, memory, external APIs.
- Always:
  - Read the code first.
  - Run tests before committing.

Prompt Patterns: Larger Changes

- For bigger tasks, structure prompts:
  1. Describe the current state (files, responsibilities).
  2. Describe the desired state (new behavior).
  3. Ask for a **plan** first.
  4. Then ask for diffs per file.
- Example:
  - "Given this backend, add JWT-based auth. First, propose a plan. Then show patches file by file, including tests."
- Keep the loop tight:
  - Apply one patch, run tests, then iterate.

Example Workflow: Small Web Service

- Step 1: Ask AI to design a simple REST API for a TODO list.
- Step 2: Generate initial server code (e.g., Python + FastAPI, Node + Express).
- Step 3: Ask AI to:
  - Add persistence (in-memory, then database).
  - Generate unit tests and a simple client example.
- Step 4: Use AI to:
  - Add error handling and input validation.
  - Suggest logging and observability hooks.
- Step 5: Manually review endpoints and tests before deployment.

Tests and Documentation with AI

- Use AI early to:
  - Generate unit and integration tests for critical paths.
  - Propose edge cases you might have missed.
- Ask AI to write:
  - Function docstrings and module overviews.
  - Developer onboarding guides and API docs.
- Treat tests as:
  - **Backstops** for future AI-generated changes.
  - Documentation of expected behavior and constraints.

Debugging AI-Generated Code: Workflow

- When something breaks:
  1. Reproduce the bug with a small, clear example.
  2. Localize the problem: which file, function, or API call?
  3. Ask AI for a **hypothesis**, not immediate code:
     - "Explain why this test fails and propose possible causes."
  4. Use an interactive debugger to step through execution.
  5. Only then ask AI to suggest a patch; review carefully.
- Avoid "stacking patches" blindly; keep the codebase clean.

Typical Bugs in AI-Generated Code

- Logic errors:
  - Off-by-one, incorrect branching conditions.
- Outdated or hallucinated APIs:
  - Functions that do not exist, wrong parameters or return types.
- Error handling:
  - Swallowing exceptions, broad `catch` blocks, unclear messages.
- Security issues:
  - Unsafe input handling, missing authentication checks.
- Performance pitfalls:
  - Inefficient loops, unnecessary network calls, N+1 queries.

Tools and Strategies for Debugging

- Standard debugging:
  - IDE debuggers, breakpoints, stepping, watch expressions.
  - Logging and metric dashboards.
- AI-assisted debugging:
  - Ask AI to interpret stack traces and error logs.
  - Ask for minimal reproducing examples and regression tests.
  - Use AI to suggest assertions and invariants.
- For tricky issues:
  - Reduce the problem to a small snippet.
  - Verify each assumption explicitly, not just trust the model.

Risks in AI-Generated Code

- Studies show a large fraction of AI-generated solutions contain security flaws.
- Common vulnerabilities:
  - Injection attacks (SQL, XSS, command, prompt injection).
  - Insecure deserialization or file handling.
  - Weak authentication / authorization checks.
- Code may *look* production-ready but:
  - Misses validation and edge cases.
  - Uses unsafe defaults or deprecated APIs.
- Security must be treated as a first-class concern when using AI.

Safe Use of AI Coding Assistants

- Protect secrets and data:
  - Never paste production secrets or proprietary data into public tools.
  - Use enterprise / self-hosted deployments when needed.
- Principle of least privilege:
  - Do not give AI agents unrestricted shell or repo access.
  - Carefully review any command the agent proposes to run.
- Governance:
  - Define where AI is allowed: prototypes vs. core infra.
  - Track which files were modified by AI for auditing.

“Vibe Coding” vs Engineering Discipline

- “Vibe coding”:
  - Prompting until the app “kind of works.”
  - Weak tests, unclear architecture, security as afterthought.
- Disciplined AI engineering:
  - Clear requirements, intentional design, peer review.
  - Systematic testing, monitoring, and rollback plans.
- AI makes experimentation easier, but:
  - Do not ship experiments as production.
  - Use AI to **amplify** good practices, not replace them.

Working with Agentic AI Systems

- Agent configuration:
  - Behavior files (`AGENTS.md`, `Claude.md`, etc.).
  - Clear rules: what the agent may and may not do.
- Commands and hooks:
  - Encapsulate common operations (run tests, lint, build).
  - Pre- and post-hooks for actions like tool use or commits.
- Best practices:
  - Keep humans in the loop for major changes.
  - Regularly checkpoint with commits and CI.

AI in DevOps and Operations

- AI can assist SRE and DevOps teams by:
  - Analyzing logs, metrics, and traces to detect anomalies.
  - Building narratives of incidents with likely root causes.
  - Recommending rollback or mitigation steps.
- Metrics to track:
  - Mean-time-to-detect (MTTD) and mean-time-to-repair (MTTR).
  - Number of engineers pulled into an incident.
- Limitations:
  - Heterogeneous stacks and incomplete observability data.
  - Need robust monitoring before AI can reason effectively.

Team Practices and Code Review with AI

- Use AI as:
  - A “first-pass reviewer” for style and simple bugs.
  - A helper for writing review comments and suggestions.
- Human reviewers still own:
  - Architecture, performance, correctness, and security.
  - Alignment with team conventions and long-term maintainability.
- Consider labeling AI-generated diffs in pull requests.
- Encourage developers to explain *why* they accept or reject AI suggestions.

Using AI to Learn, Not Just to Ship

- Turn AI into an interactive tutor:
  - "Explain this function line by line."
  - "Show me equivalent code in another language."
- Learn design patterns and trade-offs:
  - "Compare event-driven vs. REST approaches for this feature."
- Practice debugging as a skill:
  - Ask AI to propose **multiple** possible root causes, not just one fix.
- The goal: each project increases your **independent** skill, even if AI helps.

AI-Ready Project Checklist

- Before coding:
  - Clear problem statement and success metrics.
  - High-level architecture and data model reviewed by a human.
- During development:
  - Tests in place and run automatically.
  - AI prompts and assumptions written down.
- Before release:
  - Manual code review focused on security and correctness.
  - Observability (logs, metrics, alerts) configured.
  - Rollback strategy defined.

Mini Case Study: Todo App with AI (Plan)

- Goal: simple multi-user Todo app (web or mobile).
- Step 1: use AI to write a short spec:
  - Entities: User, Todo item, Project.
  - Features: CRUD, due dates, filters, sharing.
- Step 2: ask AI for architecture:
  - Frontend framework, backend stack, database choice.
- Step 3: generate scaffolding for:
  - Backend API and database migrations.
  - Frontend components and routing.

Mini Case Study: Bugs and Fixes

- Example issues you might encounter:
  - Missing authorization checks when editing todos.
  - Race conditions on concurrent updates.
  - UI not updating after backend changes.
- Debugging with AI:
  - Share failing tests and stack traces.
  - Ask “What is the simplest safe fix?” and then confirm.
- Harden the app:
  - Add tests for unauthorized access and invalid input.
  - Run a static security scanner and fix findings.

Key Takeaways

- AI can dramatically accelerate:
  - Ideation, design, coding, testing, and operations.
- But:
  - AI-generated code is not automatically correct or secure.
  - You remain responsible for quality, safety, and maintainability.
- Use AI as:
  - A powerful assistant and teacher.
  - A partner in experimentation and debugging.
- Build the habit:
  - **Plan** with AI, **build** with AI, **verify** with tests, and **own** the final result.

Thank you.

How will **you** integrate AI into your next project?