Intelligent Software Engineering

Dependency Management and Implementation

Zhilei Ren

Dalian University of Technology

October 1, 2025

Outline

Outline

The Two Paths of Software Implementation

- **Third-Party Integration**: Leverage existing libraries and frameworks
- **Custom Implementation**: Build functionality from scratch
- Both approaches require sophisticated tool support
- Modern development blends both strategies strategically

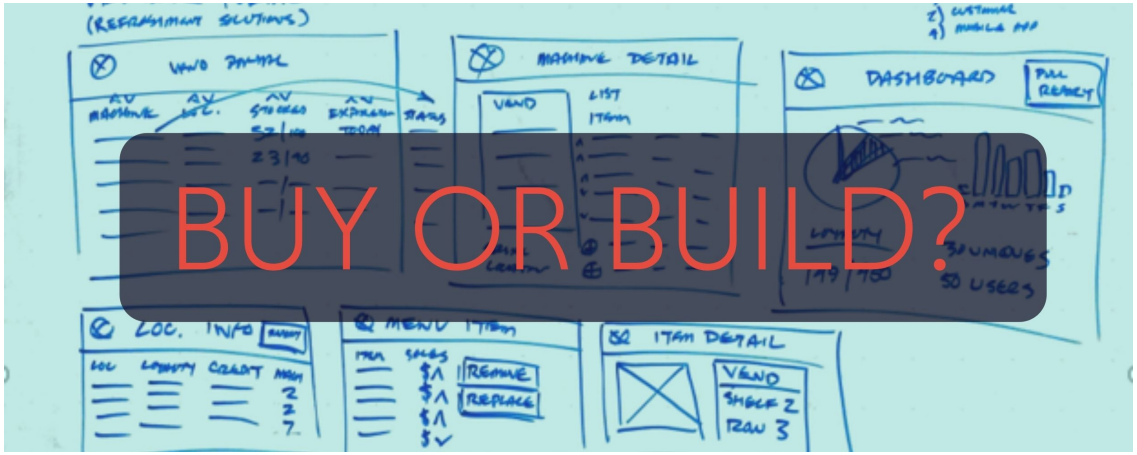


Figure 1: Third-party vs custom implementation spectrum

Dependency Management: The Integration Path

- **Challenge**: Managing external code dependencies and versions
- **Tools**: Maven, npm, pip, Gradle, NuGet
- **Key Activities**:
  - Version conflict resolution
  - Security vulnerability monitoring
  - License compliance checking
  - Build reproducibility assurance

Dependency Management: The Integration Path

- **Benefit**: Accelerated development through reuse
- **Risk**: Technical debt and security exposure

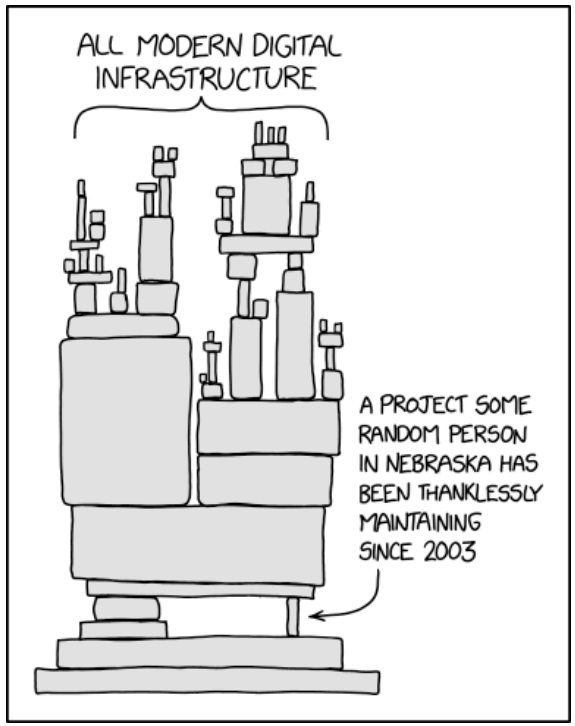


Figure 2: Dependency Risk

Code Generation: The Implementation Path

- **Challenge**: Efficiently writing and maintaining custom code
- **Tools**: LLM assistants, IDE completions, code generators
- **Key Activities**:
  - Intelligent code completion
  - API implementation generation
  - Boilerplate code automation
  - Refactoring assistance

Code Generation: The Implementation Path

- **Benefit**: Control and customization
- **Risk**: Implementation complexity and maintenance burden

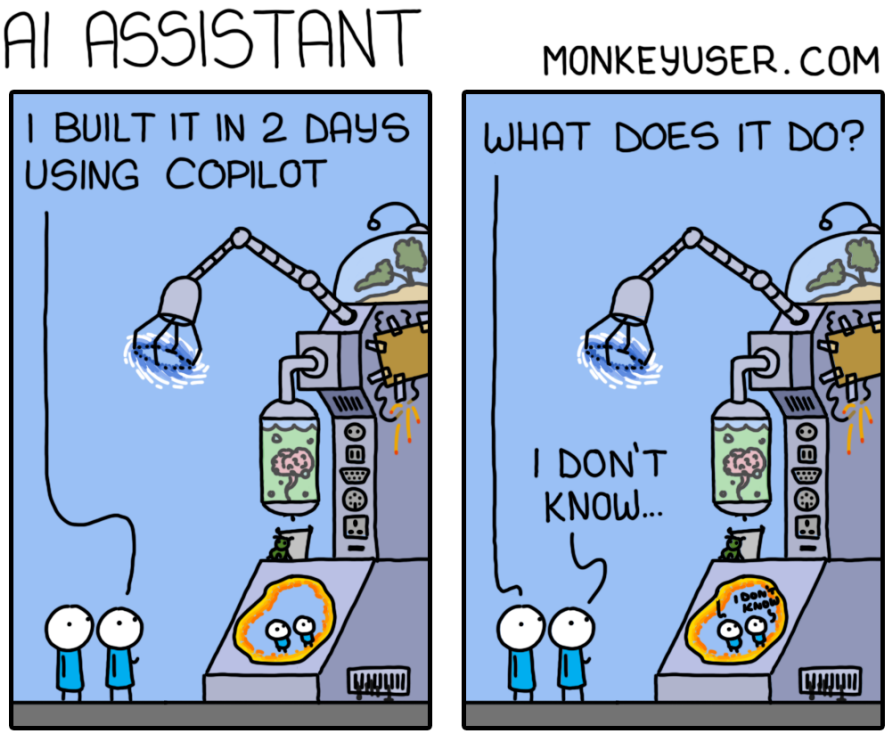


Figure 3: AI Assistant

Strategic Decision Framework

- **Choose Dependency When**:
  - Functionality is well-established and stable
  - Security and maintenance are handled by active community
  - Core competency is elsewhere
- **Choose Custom Implementation When**:
  - Functionality is a core competitive advantage
  - Special requirements not met by existing solutions
  - Long-term control and customization are critical
- **Modern Reality**: Most projects use a hybrid approach

Outline

What is Dependency Management?

Dependency management is the process of **managing external libraries, packages, and modules** that your software project relies on to function properly.

Key aspects:

- Identifying required dependencies
- Specifying version constraints
- Resolving conflicts between dependencies
- Ensuring reproducible builds
- Managing transitive dependencies

Importance

Poor dependency management can lead to **build failures, security vulnerabilities, and maintenance nightmares**.

Outline

Package Managers Overview

Package managers automate the process of **installing, upgrading, configuring, and removing software packages**.

Common package managers by ecosystem:

- **System**: apt, yum, pacman, Homebrew
- **Python**: pip, conda, Poetry
- **JavaScript**: npm, yarn, pnpm
- **Java**: Maven, Gradle
- **Ruby**: gem, Bundler
- **Rust**: Cargo

Each maintains a **central repository** of packages with versioning and dependency information.

APT - Advanced Package Tool

APT is the package management system used by **Debian and Ubuntu-based Linux distributions**.

Key commands:

- `apt update`: Refresh package lists
- `apt install <package>`: Install a package
- `apt remove <package>`: Remove a package
- `apt upgrade`: Upgrade all packages
- `apt-cache depends <package>`: Show dependencies

APT uses **Debian packages (.deb)** with metadata describing dependencies, conflicts, and recommendations.

Outline

Visualizing Dependencies with Debtree

Debtree is a tool that **generates dependency graphs** for Debian packages, helping understand complex dependency relationships.

Basic usage:

```
debtree package-name | dot -Tpng > deps.png
```

Benefits of visualization:

- Identify **circular dependencies**
- Understand **transitive dependency chains**
- Spot **unnecessary or redundant dependencies**
- Analyze **impact of package updates**

Dependency for nano

```
digraph "nano" {
        rankdir=LR;
        node [shape=box];
        "nano" -> "libncursesw6" [color=blue,label="(>= 6)"];
        "libncursesw6" -> "libtinfo6" [color=blue,label="(= 6.4+20240113-1ubuntu2)"];
        "libncursesw6" -> "libgpm2";
        "nano" -> "libtinfo6" [color=blue,label="(>= 6)"];
        "nano" -> "pico" [color=red];
        "nano" [style="setlinewidth(2)"]
        "pico" [style=filled,fillcolor=oldlace];
}
```

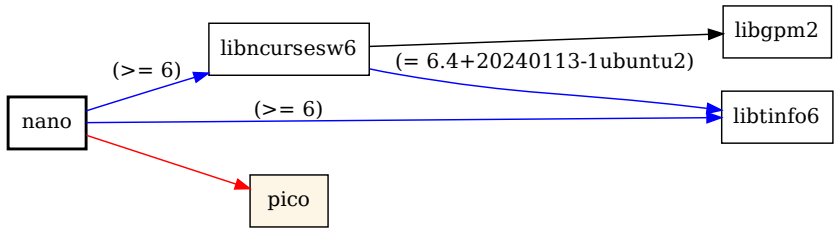


Figure 4: debtree nano | dot -T pdf > nano.pdf

Dependency for vim

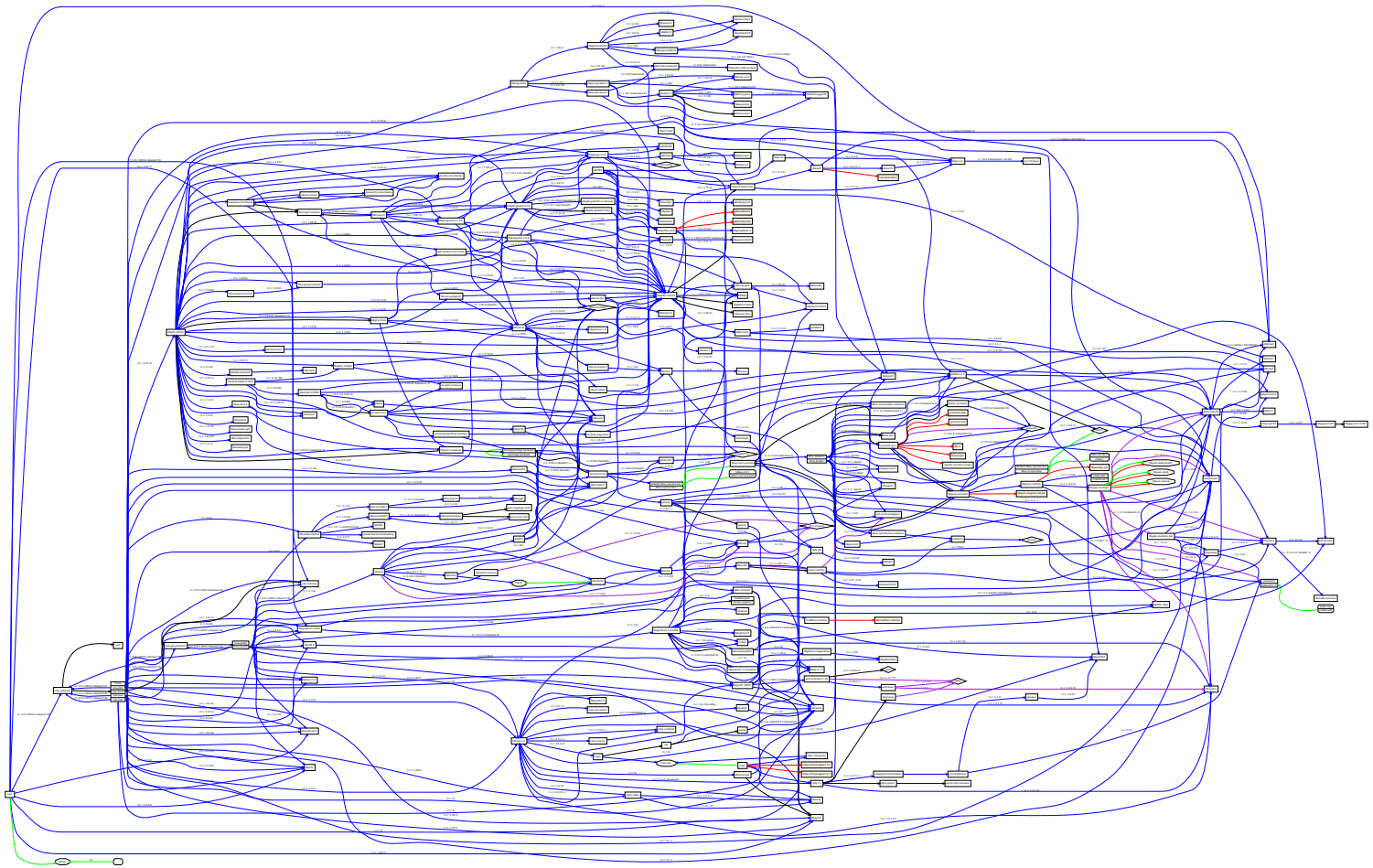


Figure 5: debtree vim | dot -T pdf > vim.pdf

Outline

Dependency Resolution Challenges

Dependency resolution is a **complex constraint satisfaction problem** that involves:

Common challenges:

- **Version conflicts**: Incompatible version requirements
- **Diamond dependency problem**: Multiple paths to same package
- **Circular dependencies**: A depends on B depends on A
- **Platform-specific dependencies**: Different requirements per OS
- **Optional dependencies**: Features that may or may not be needed

Modern package managers use **SAT/SMT/ILP solvers** to efficiently resolve these constraints.

Introduction to Z3 Theorem Prover

Z3 is a **high-performance theorem prover** developed by Microsoft Research. It's used for:

- **Constraint solving** and satisfiability checking
- **Software verification** and program analysis
- **Dependency resolution** and configuration management
- **Symbolic execution** and test case generation

Key features:

- Supports **multiple theories** (arithmetic, arrays, bit-vectors, etc.)
- Provides **Python, C++, Java, and .NET APIs**
- Used in production by Microsoft, Amazon, NASA, and others
- Can solve **complex logical constraints** efficiently

Classic Example: Chicken-Rabbit Problem

Problem Statement

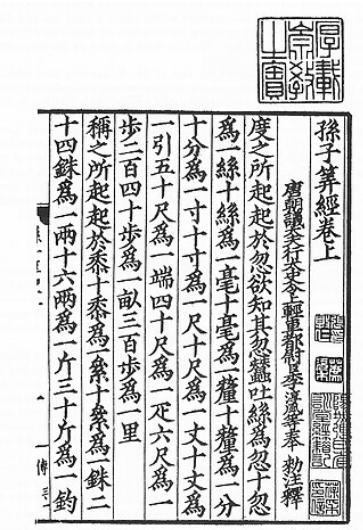
- 今有雉兔同笼，上有三十五头，下有九十四足，问雉兔各几何？
- There are chickens and rabbits in the same cage. The total number of heads is 35, and the total number of feet is 94. How many chickens and rabbits are there?
- From 孙子算经 (Mathematical Classic of Master Sun, around the 3rd-5th century AD)

Mathematical formulation:

- Let c = number of chickens, r = number of rabbits
- Constraints: $c + r = 35$ (heads) and $2c + 4r = 94$ (feet)
- Domain: $c \geq 0$, $r \geq 0$, integers

Classic Example: Chicken-Rabbit Problem

Problem Statement


孫子算經卷上

- 今有雉兔同笼，上有三十五头，下有九十四足，问
- There are chickens and rabbits in the same cage m-ber of heads is 35, and the total number of feet i ny chickens and rabbits are there?
- From 孙子算经 (Mathematical Classic of Master he 3rd-5th century AD)

Mathematical formulation:

- Let c = number of chickens, r = number of rabbits
- Constraints: $c + r = 35$ (heads) and $2c + 4r = 94$ (feet)
- Domain: $c \geq 0$, $r \geq 0$, integers

Classic Example: Chicken-Rabbit Problem

Imperative Problem Solving

- 上置三十五头，下置九十四足。半其足，得四十七。以少减多。
- Place 35 heads above and 94 feet below. Take half of the feet, which is 47. Then subtract the smaller number from the larger one.

```
num_heads = 35
num_feet = 94
num_feet_half = num_feet // 2
num_rabbits = num_feet_half - num_heads
num_chicken = num_heads - num_rabbits
```

Solving Chicken-Rabbit Problem with Z3

Declarative Problem Solving

- **What vs How**: Focus on describing *what* the problem is rather than *how* to solve it
- **Separation of Concerns**: Separate problem specification from solution algorithm
- **Examples**: SQL, Prolog, Answer Set Programming

```
import z3
chicken, rabbits = z3.Ints('chicken rabbits')
z3.solve(chicken >= 1,    # number of chicken
      rabbits >= 1,       # number of rabbits
      chicken + rabbits == 35,
      chicken * 2 + rabbits * 4 == 94)
```

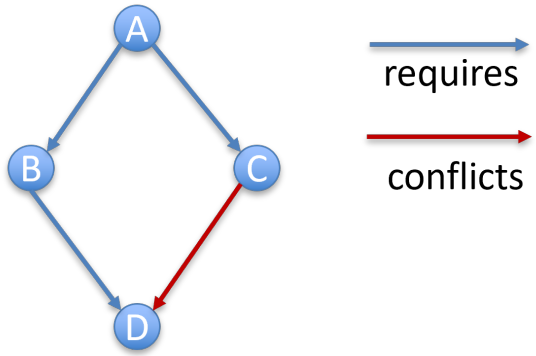
Z3 Solution and Historical Context

Connection to dependency resolution:

- Similar to resolving **version constraints** and **conflicts**
- Demonstrates how Z3 handles **integer constraints** and **equations**
- Shows the **pattern** for encoding real-world problems as constraints

This ancient problem illustrates the **fundamental principles** that modern SAT solvers like Z3 use for dependency resolution.

Encoding Dependencies to Z3




```
import z3
A, B, C, D = z3.Bools("A B C D")
p1, p2, p3, p4, p5, p6 = z3.Bools("p1 p2 p3 p4 p5 p6")
solver = z3.Solver()
solver.assert_and_track(-1 * z3.If(A, 1, 0) + z3.If(B, 1, 0) >= 0, p1)
solver.assert_and_track(-1 * z3.If(A, 1, 0) + z3.If(C, 1, 0) >= 0, p2)
solver.assert_and_track(-1 * z3.If(A, 1, 0) + z3.If(D, 1, 0) >= 0, p3)
solver.assert_and_track(-1 * z3.If(B, 1, 0) + z3.If(D, 1, 0) >= 0, p4)
solver.assert_and_track(z3.If(C, 1, 0) + z3.If(D, 1, 0) <= 1, p5)
solver.assert_and_track(A == True, p6)

print(solver.check())
print(solver.unsat_core())
```

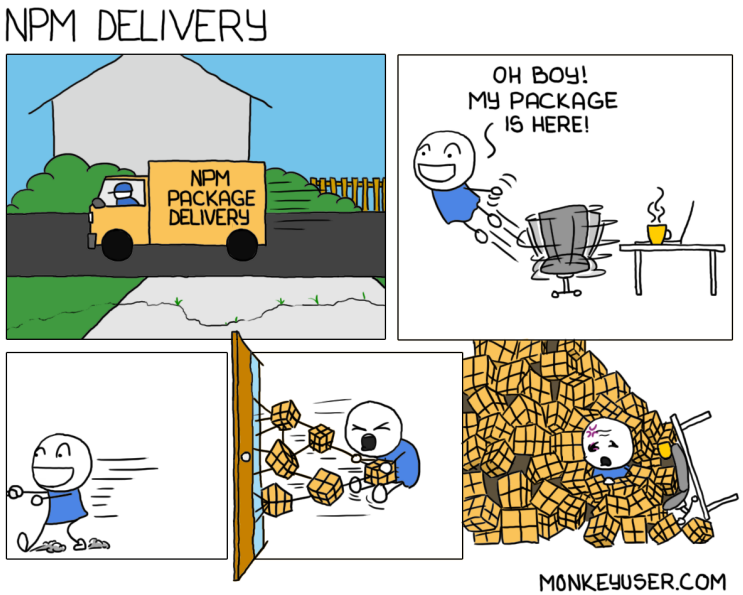
Outline

Real-World Dependency Complexity

Modern software projects often have **complex dependency graphs** with hundreds or thousands of packages.

Statistics from large projects:

- Average npm package: 75+ dependencies
- Typical web application: 1000+ transitive dependencies
- Debian: 50,000+ packages with complex inter-dependencies



Dependency Vulnerabilities

Dependencies can introduce **security vulnerabilities** that affect your application.

Common issues:

- Using outdated packages with known vulnerabilities
- Transitive dependencies with security issues
- Malicious packages in public repositories
- License compliance violations

The Log4Shell Vulnerability (CVE-2021-44228)

What is Log4Shell?

- A critical zero-day vulnerability discovered in December 2021.
- Affects Log4j, a popular, ubiquitous Java logging library.
- Affected thousands of applications and services globally, including those from major cloud providers and enterprise software vendors.

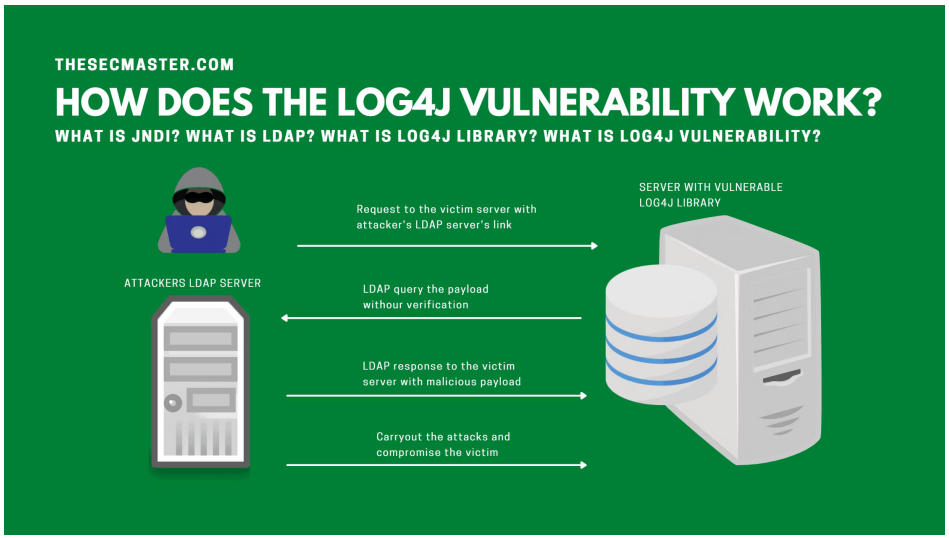


Figure 7: How Log4Shell Works

Outline

Dependency Management Best Practices

Version specification:

- Use **semantic versioning** (SemVer)
- Prefer **pinned versions** in production
- Implement **version ranges** carefully
- Maintain **lock files** for reproducibility

Development workflow:

- **CI/CD integration** for dependency checks
- **Peer review** for new dependencies
- **Documentation** of dependency choices

Modern Tools and Techniques

Advanced techniques:

- **Dependency vendoring**: Including dependencies in source
- **Reproducible builds**: Ensuring consistent artifacts
- **Supply chain security**: Verifying package integrity
- **AI-assisted dependency resolution**

Future Trends

Increased focus on **software supply chain security** and **automated dependency maintenance**.

Outline

Key Takeaways

- Dependency management is **essential for modern software development**
- Package managers like **apt automate dependency resolution**
- Tools like **debtree help visualize complex dependency graphs**
- **SAT solvers like Z3** can encode and solve dependency constraints
- **Security and maintenance** are critical aspects of dependency management
- **Best practices and automation** reduce risks and overhead

Remember

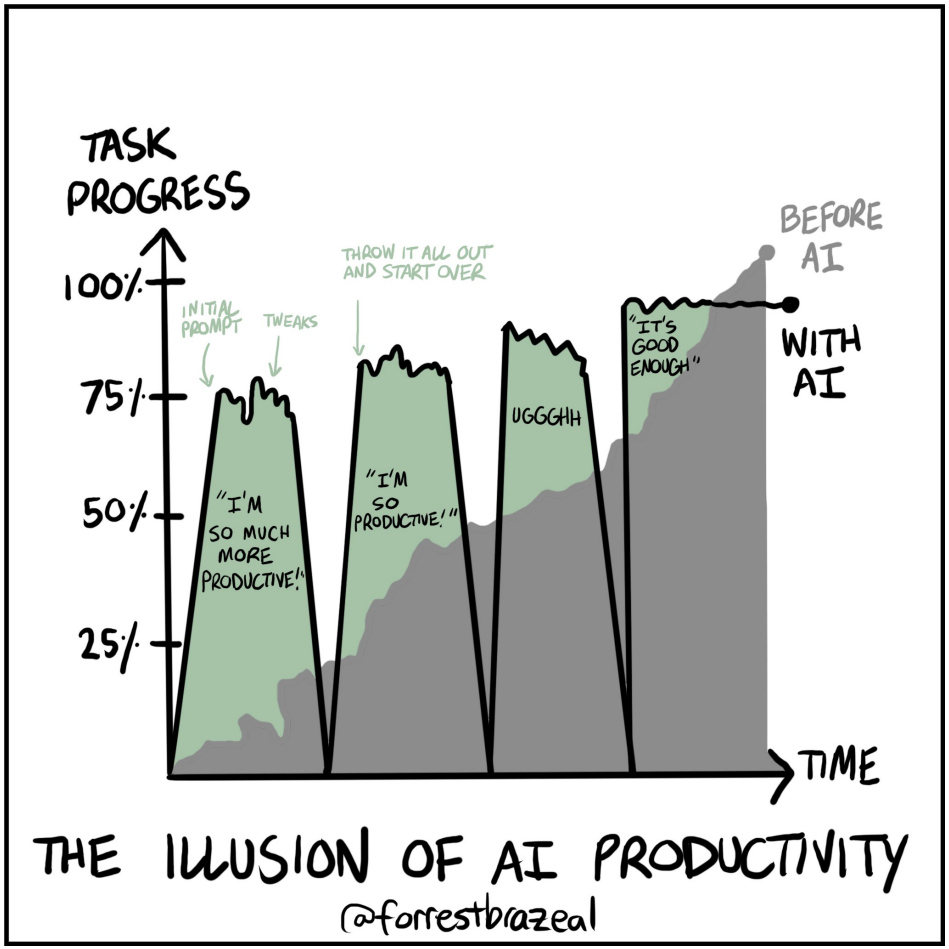
Every dependency is a **potential point of failure** - choose and manage them wisely!

Vibe Coding

Vibe coding is an artificial intelligence-assisted software development style popularized by Andrej Karpathy in February 2025. The term was listed in the Merriam-Webster Dictionary the following month as a "slang & trending" term. It describes a chatbot-based approach to creating software where the developer describes a project or task to a large language model (LLM), which generates code based on the prompt. The developer evaluates the result and asks the LLM for improvements. Unlike traditional AI-assisted coding or pair programming, the human developer avoids micromanaging the code, accepts AI-suggested completions liberally, and focuses more on iterative experimentation than code correctness or structure[1].

[1] https://en.wikipedia.org/wiki/Vibe_coding

The Illusion of AI Productivity



Outline

Intelligent Software Engineering

- **Beyond Traditional Methods**: AI and optimization techniques in software development
- **Multiple Approaches**: Different intelligent methods for different problems
- **Complementary Strengths**: Each method excels in specific domains
- **Practical Applications**: Real-world tools and techniques used today
- **Human-AI Collaboration**: Augmenting developer capabilities

Outline

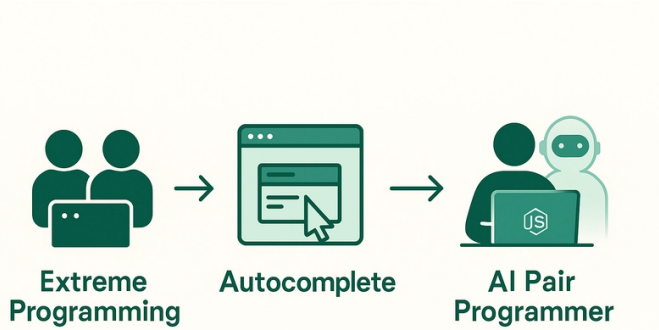
What is AI Pair Programming?

Definition

AI Pair Programming is a software development practice where developers collaborate with AI assistants to write code, receiving real-time suggestions, reviews, and optimizations.

Key Characteristics:

- **Real-time Collaboration**: AI provides instant code suggestions
- **Knowledge Sharing**: AI transfers best practices and patterns
- **Quality Assurance**: Immediate code review and optimization
- **Learning Acceleration**: Opportunity for beginners to learn from experts



What is Extreme Programming?

Core Idea

Extreme Programming (XP) is an agile software development methodology designed to improve software quality and responsiveness to changing customer requirements. It emphasizes adaptability over predictability.

Key Characteristics

- Embraces requirement changes, even late in the development cycle.
- Promotes frequent releases in short development cycles.
- Intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Core Values and Practices

Five Core Values

- **Communication**: Constant flow of information within the team and with the customer.
- **Simplicity**: Focus on the simplest solution that works today.
- **Feedback**: Rapid feedback loops from the system, customer, and team.
- **Courage**: To refactor code, throw away obsolete code, and adapt to changes.
- **Respect**: Among all team members, recognizing each contribution.

Selected Core Practices

- **Test-Driven Development (TDD)**: Write tests before code.
- **Pair Programming**: Two programmers work together at one workstation.
- **Continuous Integration**: Integrate and test code multiple times a day.

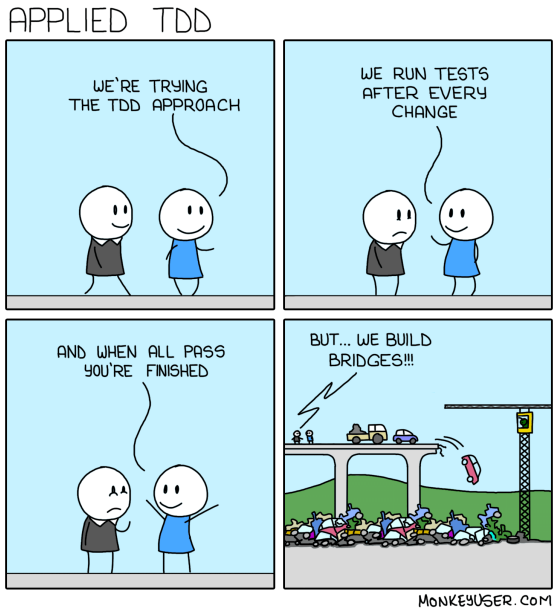
Practice in Focus: Test-Driven Development (TDD)

TDD is a relies on a very short, repetitive cycle:

1. Write a failing **automated test case**.
2. Write the minimal **code** to make the test pass.
3. **Refactor** the code to improve its structure.

Benefit

This cycle ensures nearly 100% test coverage and helps create a simple, robust design. The tests act as a safety net for future changes.



Pair Programming: Two Heads, One Keyboard

What is Pair Programming?

A core XP practice where **two programmers** work together at **one workstation**. One acts as the **Driver** (writing code) while the other serves as the **Navigator/Observer** (reviewing code and thinking strategically).

Roles

- **Driver**: Hands on keyboard, focuses on tactical implementation
- **Navigator**: Strategic thinker, reviews each line, considers alternatives
- **Roles switch frequently** (every 30-60 minutes)

Figure 9: Pair Programming Setup

Pair Programming

Key Benefits

- **Higher Code Quality**: Continuous code review reduces defects
- **Knowledge Sharing**: Prevents knowledge silos and "bus factor"
- **Better Design**: More discussion leads to better solutions
- **Training**: Junior developers learn from seniors naturally

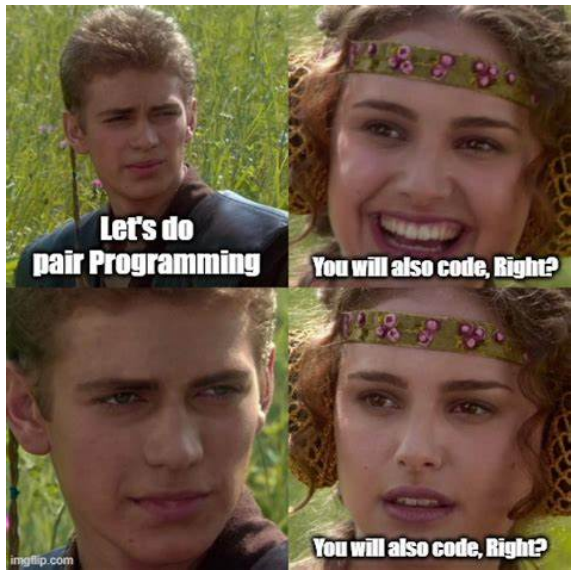


Figure 10: Actual Case

Traditional vs. AI Pair Programming

Aspect	**Traditional**	**AI Pair Programming**
Availability	Requires another developer	Available 24/7
Knowledge	Limited to developers' experience	Vast best practices coverage
Consistency	Varies by individuals	Highly consistent standards
Cost	Two developers' time	Lower tool costs
Learning	Bidirectional knowledge transfer	Primarily learning from AI
Creativity	Mutual brainstorming	Pattern-based suggestions

Table 1: Comparison of Pair Programming Approaches

Popular AI Pair Programming Tools

IDE-Integrated Tools:

- **GitHub Copilot**
  - Most popular AI programming assistant
  - Multi-language support
  - Context-aware code generation
- **Amazon CodeWhisperer**
  - AWS-optimized integration
  - Security scanning features
  - Free for individual use

Other Tools:

- **Tabnine**
  - On-premise deployment options
  - Privacy-focused
- **Cursor**
  - AI-first code editor
  - Deep code understanding
- **Replit AI**
  - Cloud development environment
  - Real-time collaboration

Outline

Introduction to AI-Assisted Programming

What is AI-Assisted Programming?

AI-assisted programming uses artificial intelligence, particularly Large Language Models (LLMs), to help developers with various coding tasks, from code completion to full project generation .

The Fundamental Shift

- Traditional programming: Writing detailed instructions (how)
- AI-assisted programming: Describing desired outcomes (what)
- This represents a move from **instruction-based** to **intent-based** programming

Key Benefit

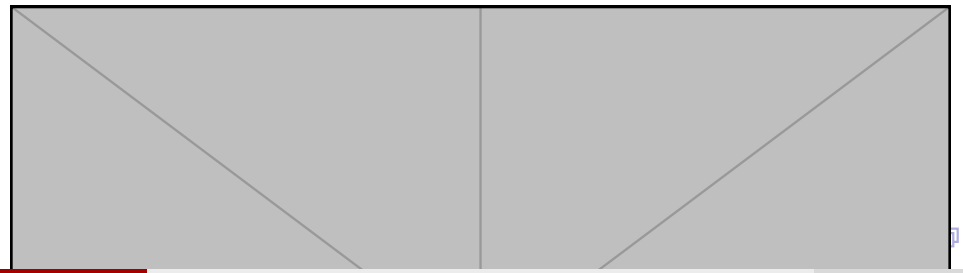
“The major benefit of AI in coding is reducing the search time by a lot, taking over routine chores, producing code drafts immediately”.

Core Technical Architecture

AI-assisted programming systems typically consist of several interconnected layers:

Layered Architecture

1. **Brain & Cognitive Core**: Powerful LLMs (GPT-4, Claude, Gemini) for code generation and reasoning
2. **Agentic Frameworks**: Systems for planning, tool use, and self-correction
3. **Execution Environment**: Safe sandboxed environments for code execution
4. **Knowledge Base**: RAG systems for project-specific context



The Agentic Workflow: Planning and Execution

AI Agent = LLM + Planning + Tools

- **LLM**: The cognitive core for understanding and reasoning
- **Planning**: Decomposing complex goals into manageable steps
- **Tools**: External APIs and functions the AI can use

ReAct Pattern: Reason + Act

AI agents follow an iterative loop:

1. **Reason**: Analyze what to do next
2. **Act**: Execute chosen action (code, call API, etc.)
3. **Observe**: Analyze results and repeat

Tool Use Capabilities

Agents can interact with file systems, version control, APIs, browsers, and execution environments.

Human-AI Collaboration in Practice

Professional AI Programming Workflow

1. **Context Preparation**: Provide all relevant project information
2. **Strategy First**: Discuss approaches before coding
3. **Select & Draft**: Choose best approach, then generate code
4. **Review & Learn**: Critical examination and understanding
5. **Test & Iterate**: Verify and refine

The "70% Problem"

AI excels at routine, pattern-based work (70% of coding tasks), but the remaining 30% - edge cases, architecture, and maintainability - still requires human expertise.

Challenges and Best Practices

Key Challenges

- **AI Hallucinations**: Confident but incorrect outputs
- **Security & Privacy**: Potential vulnerabilities in generated code
- **Intellectual Property**: Copyright concerns with training data
- **Demo-Quality Trap**: Code that works in demo but fails in production

Best Practices

- **Trust but Verify**: Never blindly trust AI-generated code
- **Iterate and Review**: Treat AI output as drafts requiring human review
- **Maintain Oversight**: Humans must remain “in the loop” for critical decisions
- **Continuous Testing**: Comprehensive testing is more crucial than

Challenges and Best Practices

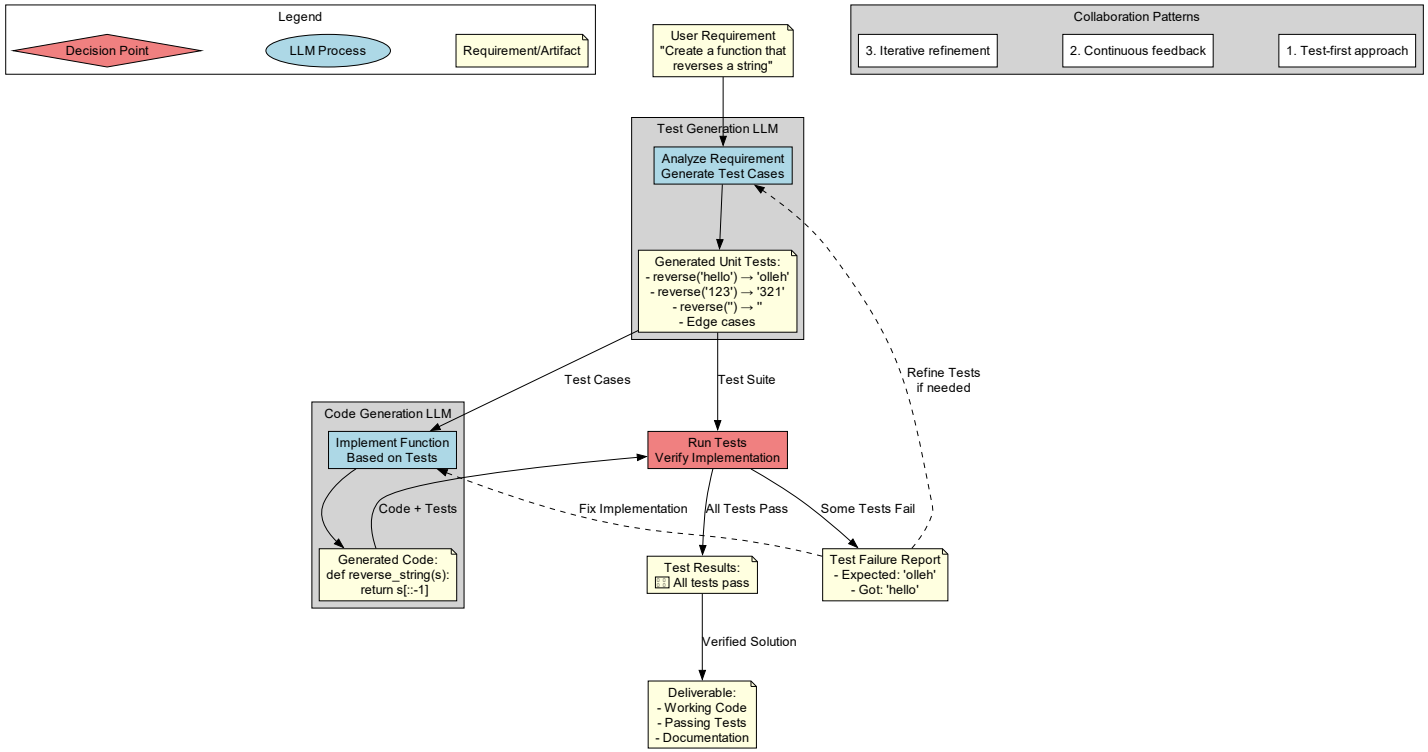


Figure 12: Test-Driven Development with LLMs

Limitations and Challenges

Current Technical Limitations

Understanding limitations helps use AI programming tools effectively.

Technical Challenges:

- **Context Length**: Limited understanding of large codebases
- **Reasoning Depth**: Difficulty with complex logic chains
- **API Knowledge**: May not know latest libraries
- **Security**: Potential for suggesting vulnerable code
- **Performance**: May suggest inefficient algorithms

Fundamental Limitations:

- **No True Understanding**: Pattern matching, not reasoning
- **Training Data Bias**: Reflects biases in training data
- **Lack of Creativity**: Cannot invent truly novel solutions
- **No Intent Understanding**: Doesn't understand why code is needed
- **Error Propagation**: Can amplify mistakes from training

Future Directions in AI-Assisted Programming

Emerging Research Areas

The field is rapidly evolving with several promising directions.

Research Frontiers:

- **Larger Context Windows**: Handling entire codebases
- **Better Reasoning**: Improved logical reasoning capabilities
- **Multimodal Understanding**: Combining code, docs, and diagrams
- **Personalization**: Adapting to individual coding styles
- **Verification Integration**: Formal verification of generated code

Architecture Innovations:

- **Retrieval-Augmented Generation**: Combining with code search
- **Program Synthesis**: Generating code from specifications

Application Areas:

- **Automated Refactoring**: Intelligent code improvement
- **Code Migration**: Porting between languages/frameworks
- **Accessibility**: Helping devel-

Outline

Function Calling: Structured AI Interactions

Beyond Text Generation

Function calling allows AI models to interact with external tools and APIs in a structured way.

What is Function Calling?

- Standardized way for models to request execution of specific functions
- Returns structured data instead of unstructured text
- Enables tool usage, API calls, and code execution
- Critical for reliable AI-assisted programming

Traditional Approach:

- Model generates text instructions
- Human interprets and executes

Function Calling Approach:

- Model requests specific function
- System executes automatically
- Structured, reliable results

Function Calling in AI Programming Assistants

Practical Applications

Function calling enables sophisticated programming workflows beyond simple code generation.

```
// Example: Function calling for code analysis
const availableFunctions = {
    analyzeSyntax: {
        description: "Analyze code syntax and structure",
        parameters: {
            code: "string",
            language: "string"
        }
    },
    runTests: {
        description: "Execute test cases on code",
        parameters: {
            code: "string",
            testCases: "array"
        }
    },
```

How Function Calling Works Technically

Architecture Overview

Function calling involves coordination between the AI model, function registry, and execution environment.

Technical Flow:

1. **Function Registration**: Available functions are defined with schemas
2. **Model Decision**: AI decides when to call functions based on context
3. **Structured Request**: Model outputs function call with parameters
4. **Execution**: System executes the function with provided parameters
5. **Result Integration**: Function results are fed back to the model

Benefits:

- **Reliability**: Structured data reduces errors

Challenges:

- **Security**: Managing execution permissions

Example: Function Calling for Code Refactoring

Real-world Workflow

Function calling enables complex multi-step programming tasks.

```
// User request: "Refactor this function to be more efficient"

// Step 1: AI analyzes the code and decides to call analysis functions
{
    "function_call": {
        "name": "analyzeComplexity",
        "parameters": {
            "code": "function processData(data) {...}",
            "metrics": ["cyclomatic", "cognitive"]
        }
    }
}

// Step 2: System returns complexity analysis
{
    "cyclomatic_complexity": 8,
    "cognitive_complexity": 12,
    "suggestions": ["Extract helper functions", "Simplify conditionals"]
}

// Step 3: AI generates refactored code and calls validation
{
    "function_call": {
        "name": "validateRefactoring",
```

Why Students Should Understand This

Career-Relevant Knowledge

Understanding these concepts prepares students for the future of software development.

Why This Matters for Students:

- **Industry Standards**: These technologies are becoming standard in professional tools
- **System Design Skills**: Understanding distributed AI systems
- **API Design Knowledge**: Learning how to design AI-friendly interfaces
- **Security Awareness**: Understanding AI system security implications

Learning Outcomes:

- Understand how modern AI programming tools work internally
- Design systems that can integrate with AI assistants
- Evaluate AI tool capabilities and limitations
- Make informed decisions about AI tool adoption

Outline

Emerging Trends in Intelligent Development

- **Automated Design Synthesis**: From requirements to implementation
- **Context-Aware Completion**: Understanding project-specific patterns
- **Real-time Design Validation**: Continuous constraint checking
- **Personalized Code Generation**: Adapting to individual coding styles
- **Multi-modal Design Tools**: Combining code, diagrams, and specifications

Outline

Effective Intelligent Design Assistance

- **Problem-Solution Fit**: Match method to design challenge characteristics
- **Incremental Adoption**: Start with well-defined subproblems
- **Validation Strategy**: Always verify intelligent system outputs
- **Human Oversight**: Maintain designer control and understanding
- **Documentation**: Record AI-assisted design decisions and rationale

Conclusion

- **Rich Methodology**: Diverse intelligent approaches for software design
- **Practical Value**: Accelerating and improving design decisions
- **Complementary Nature**: Different methods excel at different tasks
- **Human-Centric**: Augmenting rather than replacing designers
- **Rapid Evolution**: Continuous improvement in intelligent assistance

Key Insight: Intelligent methods provide powerful assistance for complex software design and implementation challenges

Thank You Questions?