Intelligent Software Engineering

Introduction to Artificial Intelligence

Zhilei Ren

Dalian University of Technology

September 26, 2025

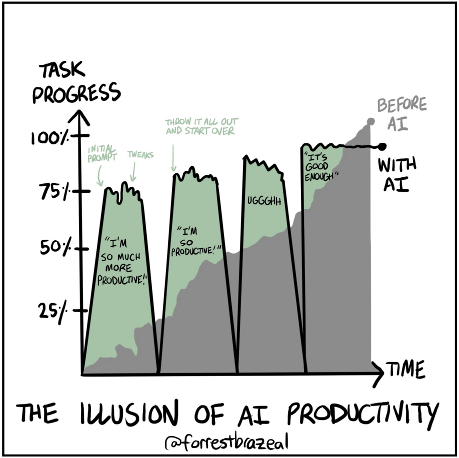
Outline

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

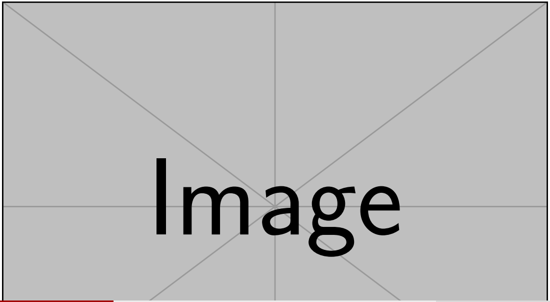


Intelligent Software Engineering

- **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

Spectrum of Intelligent Approaches

- **LLM-based Methods**: Natural language understanding and generation
- **Search-based Optimization**: Evolutionary algorithms and local search
- **Constraint Solving**: Formal methods and SAT solving
- **Hybrid Approaches**: Combining multiple intelligent techniques
- **Specialized AI**: Domain-specific machine learning models



LLM-based Approaches

- **Foundation**: Transformer architecture trained on massive text/-code corpora
- **Strengths**: Natural language understanding, code generation, documentation
- **Applications**: Code completion, architecture design, API generation
- **Examples**: GitHub Copilot, ChatGPT, CodeLlama
- **Limitations**: Hallucinations, lack of formal guarantees

LLM Code Implementation Example

```java
// Prompt: "Implement a thread-safe LRU cache in Java"
public class LRUCache<K, V> {
    private final int capacity;
    private final LinkedHashMap<K, V> cache;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new LinkedHashMap<K, V>(
            capacity, 0.75f, true) {
            protected boolean removeEldestEntry(
                Map.Entry<K, V> eldest) {
                return size() > capacity;
            }
        };
    }
```

SBSE for Software Design and Implementation

- **Foundation**: Treat software design as search problem in solution space
- **Strengths**: Finding optimal or near-optimal design solutions
- **Applications**: Algorithm selection, data structure optimization, code synthesis
- **Key Insight**: Software design decisions can be optimized systematically

SBSE for Algorithm Selection and Implementation

- **Problem**: Choose optimal algorithm for specific problem constraints
- **Search Space**: Different algorithms and their parameterizations
- **Fitness Function**: Runtime complexity, memory usage, implementation complexity
- **Method**: Genetic algorithm exploring algorithm combinations
- **Result**: Best algorithm choice with optimized parameters

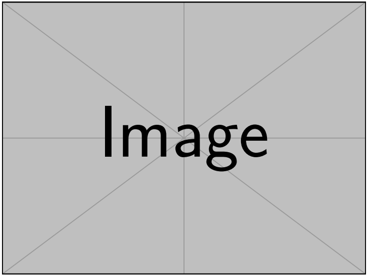


Figure 2: Algorithm selection search space

SBSE for Data Structure Optimization

```
// Problem: Optimize data structure for frequent inser
// Search-based approach explores different data struc

// Candidate solutions explored:
Solution 1: ArrayList with binary search (O(n) insert,
Solution 2: LinkedList (O(1) insert, O(n) lookup)
Solution 3: Balanced BST (O(log n) insert, O(log n) lo
Solution 4: Hash table with lazy deletion (O(1) insert

// Fitness evaluation based on actual usage patterns
// Optimal solution selected: LinkedList for 90% inser
```

SBSE for Code Synthesis and Completion

- **Challenge**: Automatically complete partial code implementations
- **Approach**: Genetic programming with code fragments as building blocks
- **Fitness**: Type correctness, test case passing, code quality metrics
- **Application**: Auto-completing complex algorithmic implementations
- **Example**: Synthesizing efficient matrix operations from specifications

SBSE for API Implementation Completion

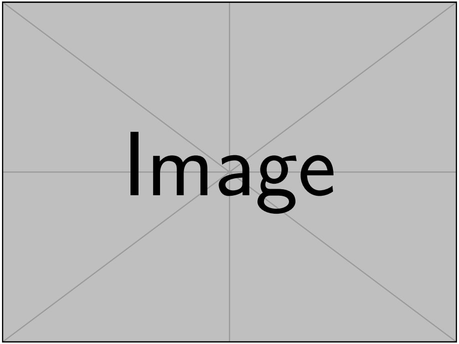
```
// Partial implementation provided by developer
public interface DataProcessor {
    Data process(Input input);
    // Additional methods to be completed...
}

// SBSE generates complete implementation based on usa
public class EfficientDataProcessor implements DataPro
    public Data process(Input input) { /* optimized */
    public void validate(Data data) { /* auto-generate
    public Result batchProcess(List<Input> inputs) { /
    // Additional methods synthesized by search-based
}
```

Constraint Solving for Software Design

- **Foundation**: Encode design constraints as logical formulas
- **Strengths**: Guaranteed satisfaction of specified properties
- **Applications**: Type-driven synthesis, interface conformance, design patterns
- **Key Benefit**: Formal verification of design decisions

Constraint-Based API Design and Implementation

- **Problem**: Ensure API consistency and completeness
- **Constraints**: Method preconditions, postconditions, invariants
- **Method**: Enforce constraints during API evolution
- **Application**: Automated checking of API design rules
- **Benefit**: Early detection of design violations



Constraint-Based Code Completion

```
// Partial code with type constraints
public <T> T process(List<T> items) {
    // Constraint solver infers missing operations
    // Constraints: T must support comparison, seriali
    // Based on usage context and method signatures

    // Solution generated by constraint solver:
    Collections.sort(items);  // T must implement Comp
    return serializer.serialize(items); // T must be s
}

// Constraint solver ensures type safety and API consi
```

Constraint-Based Design Pattern Implementation

- **Objective**: Correctly implement design patterns with formal guarantees
- **Constraints**: Pattern-specific rules (Observer: subject-observer relationships)
- **Method**: Encode pattern constraints as logical formulas
- **Verification**: Check implementation against pattern constraints
- **Completion**: Suggest missing pattern elements

Constraint-Based Singleton Pattern Enforcement

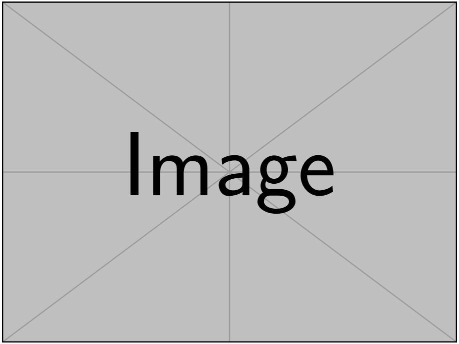
```
// Constraint: Singleton class must have private const
// and static getInstance method

class DatabaseConnection {
    private static DatabaseConnection instance;

    // Constraint solver verifies:
    //  Constructor is private
    //  getInstance method exists and is static
    //  Instance variable is static

    private DatabaseConnection() {}  // Verified: priv
    public static DatabaseConnection getInstance() { /
        if (instance == null) {
            instance = new DatabaseConnection();
```

NLP for Design Specification Processing

- **Foundation**: Extract design intent from natural language
- **Strengths**: Bridging requirement documents and implementation
- **Applications**: Design pattern recognition, architecture extraction
- **Evolution**: From manual analysis to automated understanding

Combining Methods for Design and Implementation

- **LLM + Constraints**: Generate code with formal verification
- **SBSE + Constraints**: Search with guaranteed constraint satisfaction
- **NLP + SBSE**: Extract design constraints for optimization
- **Multi-method Integration**: Comprehensive design assistance



Hybrid Example: Intelligent Code Completion

```
// Developer writes partial method:
public String processData(String input) {
    // Step 1: LLM suggests initial completion
    if (input == null) return "";
    String result = input.trim();

    // Step 2: Constraint solver verifies null safety
    //  input checked for null,  return value not null

    // Step 3: SBSE optimizes string operations
    // Replaces inefficient concatenation with StringB

    // Step 4: Final verified and optimized code
    StringBuilder sb = new StringBuilder();
    sb.append(result.toLowerCase());
```

Software Design and Implementation

- **LLMs**: Rapid prototyping and boilerplate generation
- **SBSE**: Optimal algorithm and data structure selection
- **Constraint-based**: Type-safe API design and implementation
- **Hybrid**: End-to-end design and implementation assistance

Code Completion and Synthesis

- **LLMs**: Context-aware code suggestions
- **SBSE**: Optimization-driven completion
- **Constraint-based**: Type-directed synthesis with guarantees
- **NLP**: Requirement-driven implementation

Architecture and Pattern Implementation

- **Constraint-based**: Formal pattern verification
- **SBSE**: Optimal pattern instantiation
- **LLMs**: Pattern explanation and examples
- **Hybrid**: Pattern-compliant architecture generation

Method Comparison for Design/Implementation

Method	Correctness Guarantees	Creativity	Speed
LLM-based	Low	High	Fast
SBSE	Medium	Medium	Medium
Constraint-based	High	Low	Slow
Hybrid	High	High	Medium

- **Correctness Guarantees**: Formal verification capabilities
- **Creativity**: Novel solution generation
- **Speed**: Response time for practical use

Strengths for Design and Implementation

- **LLMs**: Excellent for exploratory design and rapid prototyping
- **SBSE**: Superior for optimization problems and algorithm selection
- **Constraint-based**: Unmatched for correctness-critical components
- **Hybrid**: Balanced approach for complex design challenges

Design and Implementation Tools

- **LLM-based**: GitHub Copilot, Amazon CodeWhisperer, Tabnine
- **SBSE**: Program synthesis tools (SKETCH, Rosette)
- **Constraint-based**: Z3 for program verification, Alloy for design
- **Hybrid**: Intelligent IDEs with multiple AI assistants

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

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?