AI for Large-Scale Code Transformations: Revolutionizing Developer Workflows

Subtitle: Moving Beyond Line-by-Line Edits to Repository-Wide Intelligence

Transformations

✅ Package Migrations - Update dependencies across entire codebase

✅ Build & Bug Fixing - Automated error resolution

✅ Code Refactoring - Structural improvements maintaining behavior

✅ Fixing Code Styling Issues - Enforce standards automatically

✅ Static Analysis-Based Repairs - Fix linter/analyzer warnings

✅ Static Analysis Annotations - Add type hints, documentation

The Challenge - Why LLMs Alone Fall Short

Problem 1: Context Window Limitations

- The whole codebase cannot fit into one LLM call

- Need to break the codebase into chunks and make multiple LLM calls, each with appropriate context

- Requires automated orchestration mechanisms that understand software structure

Problem 2: Knowledge Cutoffs

- Models are limited by their training data cutoff dates, lacking awareness of newer developments such as updated library versions or frameworks.

- This limitation is particularly significant in programming, where API changes occurring after the training cutoff date are not reflected in the LLM's responses.

Problem 3: Hallucinations

- LLMs may generate syntactically valid but logically incorrect code (e.g., using non-existent APIs or ignoring edge cases).

CodePlan - Overview

CodePlan: Repository-Wide Intelligent Planning

What it does:

Automates large, interdependent code changes—such as package migrations or multi-file refactorings—by treating the problem as a planning task.

Key Innovation:

Frames repository-level coding as a planning problem, not just a generation problem.

Example Use Cases:

- Package migration (e.g., C# package updates across 2-97 files)
- Temporal code edits (Python dependency updates)
- Static analysis-based repairs
- Type annotation additions

CodePlan - Overview

What it is:

- A neuro-symbolic framework for building AI-powered software development workflows for inner/outer loops of software engineering

Core Philosophy:

- Integrates the power of LLMs with intelligent planning for tackling complex coding tasks that cannot be solved using LLMs alone

Key Capabilities:

- Collections of modules, transforms, actions, and prompts
- Extensible for SE tasks across entire repositories
- Combines neural (LLM) and symbolic (static analysis) approaches

CodePlan - Architecture

System Components

Inputs:

- \- Code Repository/Directory
- \- Knowledge Base Repository/Instructions
- \- Static analysis reports and logs

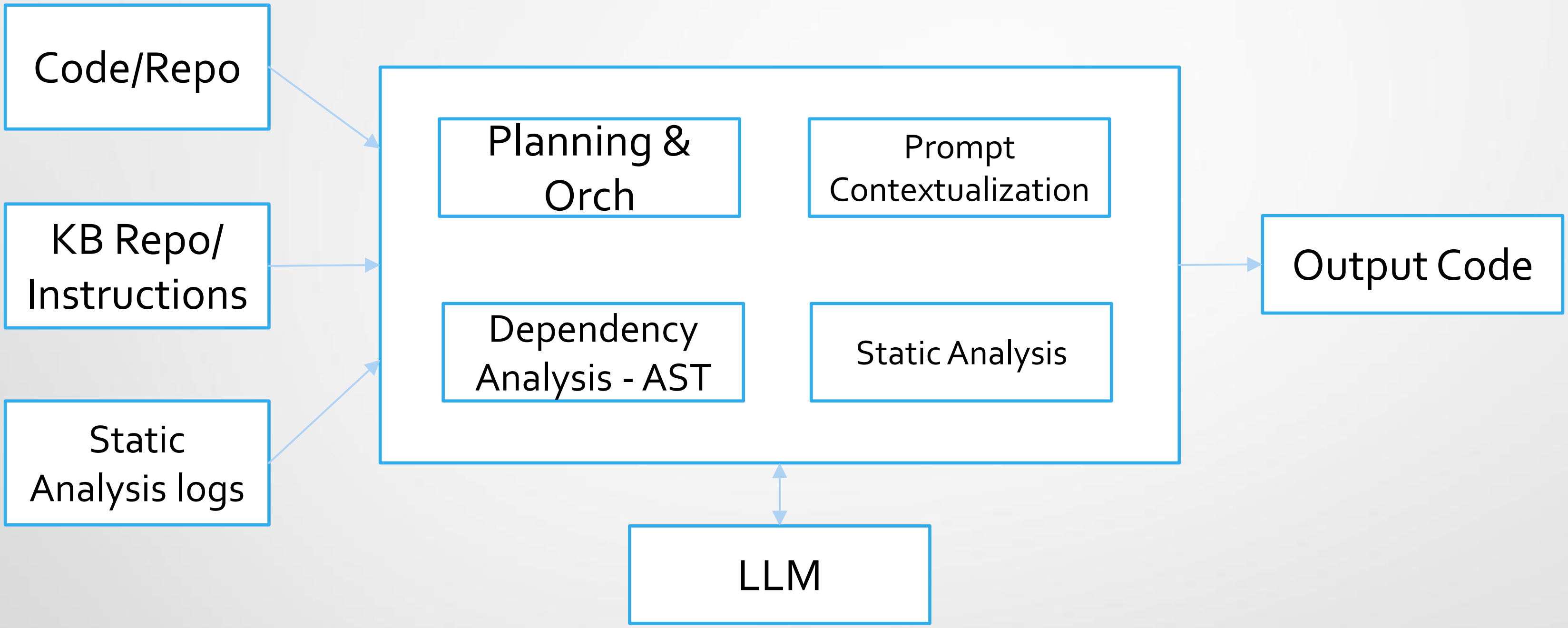
CodePlan Engine:

- 1. Prompts Contextualization - Prepares LLM prompts with relevant context
- 2. Planning & Orchestration - Coordinates multi-step transformations
- 3. Dependency Analysis - Uses AST for code structure understanding
- 4. Static Analysis - Validates and guides transformations

Output:

- \- Generated/transformed code

Architecture



Planning + AST-Based Chunking + RAG

LLM Planning

- Breaks complex tasks into **structured steps**
- Chooses **tools, order, and reasoning path**
- Enables **goal-directed execution** instead of one-shot answers

AST-Based Chunking

- Splits code/documents using **Abstract Syntax Trees**
- Preserves **semantic boundaries** (functions, classes, loops)
- Produces **syntactically valid chunks** → better embeddings & retrieval

Retrieval-Augmented Generation (RAG)

- **Retriever**: finds relevant chunks from knowledge base
- **Generator (LLM)**: integrates retrieved context into responses
- Reduces **hallucinations**, overcomes **context limits**, adds **fresh knowledge**

AST Structure Example

From Code to AST

Example Python Code:

```python
def greet(name):
    return "Hello, " + name
```

Its AST Representation:

```
FunctionDef
├── Name: greet
├── Arguments: name
└── Return
    └── BinaryOp (+)
        ├── String: "Hello, "
        └── Variable: name
```

What This Enables:

- Precise code understanding at syntactic level
- Relationship mapping between code components
- Meaningful code chunking for LLM processing

Abstract Syntax Trees (AST) - The Foundation

The Problem with Raw Code:

- LLMs see code as plain text, not structured logic
- Can't naturally recognize functions, loops, properties, relationships
- Results in generic, inaccurate responses based on guessing

The AST Solution:

- Breaks code into organized, hierarchical pieces
- Allows AI to understand methods, properties, arguments, data types, and logic
- Makes AI responses more accurate and insightful

Traditional vs AST-Based Chunking

The Problem with Traditional Chunking

Traditional Approach (Character-Based):

```
Chunk 1: #include <iostream>
       using namespace std;
Chunk 2: void greet(string name) {
Chunk 3: cout << "Hello, " << name << endl;
       }
```

❌ Problems:

- Disregards code structure
- Produces malformed fragments
- Lacks proper syntax closure
- Breaks semantic meaning

AST-Based Chunking Solution

AST-Based Chunking: A Better Approach

Using Tree-sitter Parser:

Result:

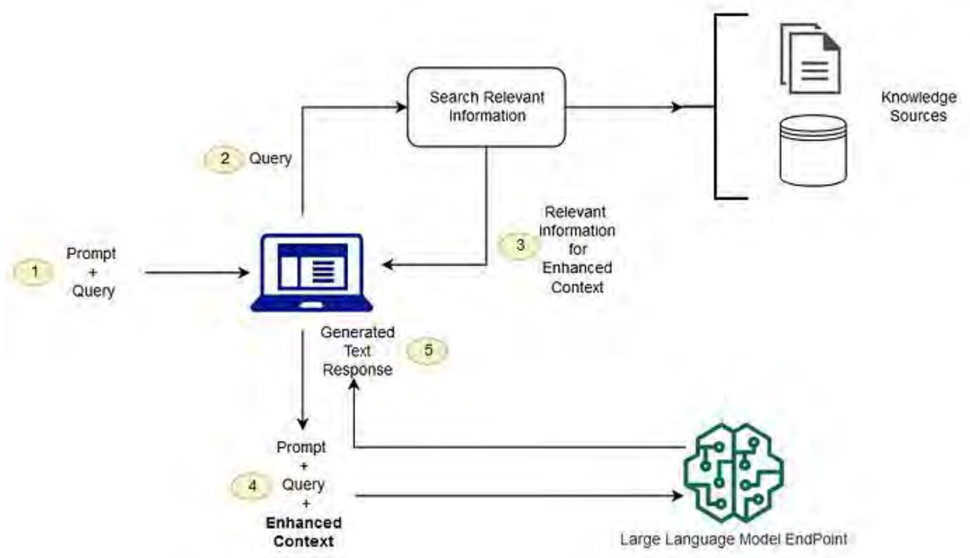
```
Chunk 1: preproc_include → #include <iostream>
Chunk 2: using_declaration → using namespace std;
Chunk 3: function_definition → void greet(string name) {
    cout << "Hello, " << name << endl;
  }
Chunk 4: function_definition → int main() {
    greet("Alice");
    greet("Bob");
    return 0;
  }
```

✅ Benefits:

- Splits code at meaningful boundaries (functions, control structures)
- Each chunk remains syntactically valid
- Preserves semantic integrity
- Better LLM comprehension and code generation

RAG

- A **retriever** that searches and extracts relevant information from knowledge sources
- A **generator** that processes this information through LLMs to produce refined outputs



Real-World Impact

Time Savings:

- 70% reduction in modernization timelines (mainframe migration studies)

- Automate tasks that previously took 3-5 years

Quality Improvements:

- 98.1% functional equivalence maintained in transformations

- Reduced manual errors in large-scale refactoring's

Developer Productivity:

- Frees developers from repetitive, error-prone tasks

- Allows focus on high-value architecture and design work

- Enables non-experts to perform complex code transformations

Enterprise Applications:

- Package migrations across hundreds of files

- Legacy code modernization

- Compliance and security updates at scale

Challenges & Future Directions

Challenges:

- Context length still limits extremely large codebases
- Model hallucinations require robust validation
- Computational cost for repository-scale analysis
- Complexity in handling dynamic languages (Python, JavaScript)

Future Research Directions:

- Sliding window strategies for encoding long code
- Re-ranking mechanisms for retrieved code chunks
- Better hybrid retrieval (BM25 + semantic search)
- Fine-tuning models on domain-specific code
- Improved neuro-symbolic integration frameworks

Emerging Trends:

- Multi-agent systems for complex transformations
- Continuous learning from developer feedback
- Integration with IDE and CI/CD pipelines

Conclusion - The Future is Repository-Scale

Key Messages

The Paradigm Shift:

- Moving from line-by-line autocomplete → repository-wide reasoning
- From simple text generation → structured planning with validation
- From pure neural → neuro-symbolic hybrid systems

Critical Enablers:

- Planning frameworks that break down complex tasks
- AST-based understanding for structural code knowledge
- Multi-context integration (spatial + temporal)
- Oracle-guided validation for correctness assurance

The Bottom Line:

AI-powered large-scale code transformations are no longer theoretical—they're achieving production-ready results on real-world repositories, transforming what's possible in software engineering.