# **CodeLve: Technical Architecture & Design Document**

## **Hybrid System with Smart Templates + CodeGen-2B**

## **📋 Executive Summary**

**Product**: CodeLve - Enterprise AI Code Assistant  
 **Architecture**: Hybrid Prompt Engineering + AI Code Analysis  
 **Models**:

* **TinyLlama-1.1B** (~2GB) - Complex Query Fallback (Optional)
* **CodeGen-2B** (2B params, ~4GB) - Code Analysis Layer **Total Size**: 4-6GB (Less than IntelliJ IDEA)  
   **Processing**: 100% Local, Zero Cloud Dependencies

## **🏗️ System Architecture Overview**

┌─────────────────────────────────────────────────────────────────────┐

│ CodeLve Desktop Application │

├─────────────────────────────────────────────────────────────────────┤

│ │

│ ┌─────────────┐ ┌──────────────┐ ┌────────────────────┐ │

│ │ Flet UI │ │ FastAPI │ │ File Scanner │ │

│ │ (Desktop) │◄──►│ Web Server │◄──►│ (Python) │ │

│ └─────────────┘ └──────────────┘ └────────────────────┘ │

│ ▲ ▲ │ │

│ │ │ ▼ │

│ │ ┌──────┴──────────┐ ┌──────────────────┐ │

│ │ │ Query Router │ │ Codebase Memory │ │

│ │ │ & Orchestrator │ │ (In-Memory) │ │

│ │ └─────────────────┘ └──────────────────┘ │

│ │ │ ▲ │

│ ▼ ▼ │ │

│ ┌──────────────────────────────────────────────────┘ │

│ │ Hybrid Processing Pipeline │

│ │ ┌─────────────────┐ ┌─────────────────────────┐ │

│ │ │ Smart Templates │ │ CodeGen-2B │ │

│ │ │ (0 MB - Code) │─────►│ Code Analyzer │ │

│ │ └────────┬────────┘ │ (4GB - Model) │ │

│ │ │ Fallback └─────────────────────────┘ │

│ │ ▼ │

│ │ ┌─────────────────┐ │

│ │ │ TinyLlama │ │

│ │ │ (2GB Optional) │ │

│ │ └─────────────────┘ │

│ └────────────────────────────────────────────────────────────── │

│ │

└─────────────────────────────────────────────────────────────────────┘

## **🔄 Data Flow Architecture**

User Query Flow:

═══════════════

[User Input]

│

▼

[Query Router]

│

├─► [Query Type Detection]

│ │

│ ├─► Simple Query ──► [Smart Templates]

│ │ │

│ └─► Complex Query ─► [TinyLlama]

│ │

│ ▼

└─► [Context Retrieval] ◄─── [Codebase Memory]

│

▼

[Optimized Prompt Generation]

│

▼

[CodeGen-2B Analysis]

│

├─► Code understanding

├─► Pattern recognition

├─► Business logic extraction

│

▼

[Response Formatter]

│

▼

[Streaming Response to UI]

## **🧠 Hybrid Processing Pipeline**

### **Stage 1: Smart Prompt Engineering**

# Intelligent Prompt Optimization Flow

┌─────────────────────────────────────────┐

│ Query Classification │

├─────────────────────────────────────────┤

│ Input: "How does auth work?" │

│ │

│ Complexity Score: 0.3 (Simple) │

│ Type: COMPONENT\_ANALYSIS │

│ Confidence: 0.95 │

└───────────────┬─────────────────────────┘

│

├─[Simple]──► Smart Templates

│ (Instant, 0ms)

│

└─[Complex]─► TinyLlama

(Fast, ~500ms)

┌─────────────────────────────────────────┐

│ Smart Template Processing │

├─────────────────────────────────────────┤

│ 1. Pattern matching (Regex/NLP) │

│ 2. Entity extraction │

│ 3. Context injection │

│ 4. Structure formatting │

│ │

│ Output: Optimized prompt in <5ms │

└─────────────────────────────────────────┘

### **Stage 2: Code Analysis (CodeGen-2B)**

# Code Analysis Flow (Unchanged)

┌─────────────────────────────────────────┐

│ Enhanced Prompt + Context │

├─────────────────────────────────────────┤

│ - Optimized query (from templates) │

│ - Relevant code snippets │

│ - File relationships │

│ - Business context │

└───────────────┬─────────────────────────┘

│

▼

┌─────────────────────────────────────────┐

│ CodeGen-2B Processing │

├─────────────────────────────────────────┤

│ - Code parsing & understanding │

│ - Pattern recognition │

│ - Logic flow analysis │

│ - Dependency mapping │

│ - Security assessment │

└───────────────┬─────────────────────────┘

│

▼

┌─────────────────────────────────────────┐

│ Structured Analysis Output │

└─────────────────────────────────────────┘

## **📁 Smart Template System**

### **Template Engine Architecture**

class SmartPromptEngine:

"""

Deterministic prompt optimization with intelligent fallback

"""

def \_\_init\_\_(self):

self.templates = self.\_load\_templates()

self.query\_classifier = QueryClassifier()

self.entity\_extractor = EntityExtractor()

self.tinyllama = None # Lazy loaded only if needed

def optimize\_prompt(self, query: str, context: dict) -> str:

"""

Main entry point for prompt optimization

"""

# 1. Classify query

query\_type, confidence = self.query\_classifier.classify(query)

# 2. Extract entities

entities = self.entity\_extractor.extract(query)

# 3. Decide processing path

if confidence > 0.8 and query\_type in self.templates:

# Use deterministic templates (95% of queries)

return self.\_apply\_template(query\_type, entities, context)

else:

# Complex query - use TinyLlama (5% of queries)

return self.\_llm\_optimize(query, context)

def \_apply\_template(self, query\_type: str, entities: dict, context: dict) -> str:

"""

Apply smart templates with entity injection

"""

template = self.templates[query\_type]

# Intelligent placeholder replacement

prompt = template.format(

\*\*entities,

files=self.\_format\_file\_list(context['files']),

code\_snippets=self.\_format\_code\_snippets(context['snippets']),

relationships=self.\_format\_relationships(context['deps'])

)

return prompt

def \_llm\_optimize(self, query: str, context: dict) -> str:

"""

Fallback to TinyLlama for complex queries

"""

if self.tinyllama is None:

self.tinyllama = self.\_load\_tinyllama()

# Use TinyLlama for complex prompt engineering

return self.tinyllama.generate(query, context)

### **Query Classification System**

class QueryClassifier:

"""

Fast, deterministic query classification

"""

def \_\_init\_\_(self):

self.patterns = {

'COMPONENT\_ANALYSIS': [

r'how does (\w+) work',

r'explain (\w+) component',

r'analyze (\w+) module'

],

'FUNCTION\_ANALYSIS': [

r'what does (\w+) function do',

r'explain function (\w+)',

r'analyze method (\w+)'

],

'ARCHITECTURE': [

r'system architecture',

r'how is .\* structured',

r'project organization'

],

'BUG\_FINDING': [

r'find bug',

r'why .\* failing',

r'debug (\w+)'

],

'CODE\_GENERATION': [

r'create (\w+)',

r'implement (\w+)',

r'write code for'

]

}

def classify(self, query: str) -> Tuple[str, float]:

"""

Returns (query\_type, confidence)

"""

query\_lower = query.lower()

for query\_type, patterns in self.patterns.items():

for pattern in patterns:

if re.search(pattern, query\_lower):

return query\_type, 0.95

# Use keyword matching for lower confidence

return self.\_keyword\_classify(query\_lower)

### **Template Library**

class TemplateLibrary:

"""

Comprehensive template collection

"""

TEMPLATES = {

'COMPONENT\_ANALYSIS': """

Analyze the {component\_name} component in the codebase:

CONTEXT FILES:

{files}

ANALYSIS REQUIREMENTS:

1. Component Architecture:

- Overall structure and design

- Key classes and their responsibilities

- Design patterns used

2. Core Functionality:

- Primary purpose

- Main operations

- Business logic implementation

3. Dependencies:

- Internal dependencies: {internal\_deps}

- External libraries: {external\_deps}

- Database/API connections

4. Data Flow:

- Input processing

- Data transformations

- Output generation

5. Integration Points:

- How other components interact with this

- API endpoints exposed

- Events triggered/consumed

6. Code Quality:

- Error handling approach

- Logging and monitoring

- Test coverage

RELEVANT CODE SNIPPETS:

{code\_snippets}

COMPONENT RELATIONSHIPS:

{relationships}

""",

'FUNCTION\_ANALYSIS': """

Analyze the function: {function\_name}

LOCATION: {file\_path}:{line\_number}

FUNCTION SIGNATURE:

{function\_signature}

FUNCTION BODY:

{function\_code}

CALLING CONTEXT:

- Called by: {callers}

- Calls: {callees}

ANALYSIS TASKS:

1. Purpose: What business logic does this implement?

2. Parameters: Explain each parameter's role

3. Return Value: What does it return and when?

4. Side Effects: Database writes, API calls, state changes

5. Error Cases: What can go wrong and how is it handled?

6. Performance: Time/space complexity, optimization opportunities

7. Security: Input validation, authorization checks

8. Testing: How to test this function effectively

RELATED CODE:

{related\_code}

""",

'ARCHITECTURE': """

Project Architecture Analysis

PROJECT: {project\_name}

SIZE: {file\_count} files, {line\_count} lines

TECHNOLOGIES: {tech\_stack}

DIRECTORY STRUCTURE:

{directory\_tree}

ANALYZE:

1. Overall Architecture Pattern (MVC, microservices, etc.)

2. Component Organization and Boundaries

3. Data Flow Between Components

4. External System Integrations

5. Security Architecture

6. Scalability Considerations

7. Technical Debt Areas

8. Recommended Improvements

KEY COMPONENTS:

{key\_components}

DEPENDENCY GRAPH:

{dependency\_graph}

""",

'BUG\_FINDING': """

Debug Analysis Request

ERROR DESCRIPTION: {error\_description}

ERROR LOCATION: {error\_location}

ERROR TYPE: {error\_type}

STACK TRACE:

{stack\_trace}

CONTEXT FILES:

{relevant\_files}

INVESTIGATION TASKS:

1. Root Cause Analysis

2. Code Path to Error

3. Data State at Error Point

4. Similar Patterns in Codebase

5. Potential Fixes

6. Impact Assessment

7. Test Cases to Add

RECENT CHANGES:

{recent\_changes}

RELATED ISSUES:

{related\_issues}

"""

}

## **📊 Performance Optimization**

### **Memory Management (Improved)**

┌─────────────────────────────────────────────┐

│ Memory Layout │

├─────────────────────────────────────────────┤

│ Models (4-6GB): │

│ - CodeGen-2B: 4GB (always loaded) │

│ - TinyLlama: 2GB (lazy loaded) │

│ │

│ Runtime (1-2GB): │

│ - Inference buffers: 0.5-1GB │

│ - Codebase index: 0.5GB │

│ - Template cache: 10MB │

│ │

│ Total: 5-8GB RAM (vs 12-14GB before) │

└─────────────────────────────────────────────┘

### **Query Processing Performance**

Query Type | Template | TinyLlama | CodeGen-2B | Total

------------------------|-----------|-----------|------------|-------

Simple (80% of queries) | 5ms | - | 2-3s | ~3s

Complex (20%) | - | 500ms | 2-3s | ~3.5s

Average Response Time: 3.1 seconds (excellent for local AI)

## **🚀 Deployment Architecture**

### **Installation Flow**

1. Download CodeLve Installer (50MB)

│

▼

2. Core Installation

├─► Install Python runtime

├─► Install dependencies

└─► Setup file structure

│

▼

3. Model Download (Progressive)

├─► Download CodeGen-2B (4GB) - Required

└─► TinyLlama (2GB) - Downloaded on first complex query

│

▼

4. Ready to Use (Basic: 4GB, Full: 6GB)

### **File System Layout**

~/.codelve/

├── models/

│ ├── codegen-2b/ # Always present

│ │ ├── model.safetensors

│ │ ├── config.json

│ │ └── tokenizer/

│ └── tinyllama/ # Downloaded when needed

│ ├── model.safetensors

│ ├── config.json

│ └── tokenizer/

├── templates/

│ ├── prompts.json # Template library

│ └── patterns.json # Query patterns

├── cache/

│ └── responses/

└── config/

└── settings.json

## **🔧 Implementation Details**

### **Query Router Implementation**

class QueryRouter:

"""

Routes queries through optimal processing path

"""

def \_\_init\_\_(self, codebase\_memory, model\_manager):

self.memory = codebase\_memory

self.models = model\_manager

self.prompt\_engine = SmartPromptEngine()

self.stats = QueryStats() # Track performance

async def process\_query(self, user\_query: str) -> str:

start\_time = time.time()

# 1. Get relevant context

context = self.memory.get\_context(user\_query)

# 2. Optimize prompt (templates or TinyLlama)

optimized\_prompt = self.prompt\_engine.optimize\_prompt(

user\_query, context

)

# 3. Track which path was used

template\_used = not self.prompt\_engine.tinyllama\_was\_used

# 4. Run CodeGen-2B analysis

analysis = await self.models.codegen.analyze(

optimized\_prompt, context

)

# 5. Format response

response = self.format\_response(analysis, user\_query)

# 6. Track stats

self.stats.record(

time.time() - start\_time,

template\_used,

len(context['files'])

)

return response

### **Entity Extraction**

class EntityExtractor:

"""

Extract relevant entities from queries

"""

def extract(self, query: str) -> dict:

entities = {}

# Function names

func\_pattern = r'function\s+(\w+)|method\s+(\w+)|def\s+(\w+)'

if match := re.search(func\_pattern, query, re.I):

entities['function\_name'] = match.group(1) or match.group(2) or match.group(3)

# Component/module names

comp\_pattern = r'(\w+)\s+(?:component|module|service|class)'

if match := re.search(comp\_pattern, query, re.I):

entities['component\_name'] = match.group(1)

# File paths

file\_pattern = r'(?:file|in)\s+([/\w\-\_.]+\.\w+)'

if match := re.search(file\_pattern, query, re.I):

entities['file\_path'] = match.group(1)

# Error messages

error\_pattern = r'error[:\s]+(.+?)(?:\.|$)'

if match := re.search(error\_pattern, query, re.I):

entities['error\_description'] = match.group(1)

return entities

## **🎯 Advantages of Hybrid Approach**

### **1. Performance**

* 95% of queries use instant templates (5ms)
* Only complex queries use TinyLlama (500ms)
* Overall 10x faster than full dual-LLM approach

### **2. Size Efficiency**

* 40% smaller than original design (6GB vs 10GB)
* TinyLlama downloaded only when needed
* Minimal base installation (4GB)

### **3. Reliability**

* Deterministic templates = consistent results
* No LLM hallucination for common queries
* Fallback ensures complex queries still work

### **4. Extensibility**

* Easy to add new templates
* Pattern matching can be enhanced
* LLM can be upgraded without breaking templates

### **5. Resource Usage**

* Lower RAM usage (5-8GB vs 12-14GB)
* Less CPU for prompt optimization
* Faster startup (only load CodeGen-2B initially)

## **🔮 Future Enhancements**

### **Short Term (3-6 months)**

1. **Template Learning**: Analyze successful TinyLlama outputs to create new templates
2. **Pattern Enhancement**: ML-based pattern matching instead of regex
3. **Context Ranking**: Better relevance scoring for context selection

### **Long Term (6-12 months)**

1. **Custom Model Training**: Fine-tune CodeGen-2B on user's codebase
2. **Template Personalization**: Learn user's query patterns
3. **Incremental Analysis**: Only analyze changed code portions
4. **Multi-Language Support**: Extend beyond Python/JS

## **📈 Success Metrics**

### **Performance Targets**

* **Template Hit Rate**: >90% of queries handled by templates
* **Average Response**: <3.5 seconds end-to-end
* **Memory Usage**: <8GB peak for 100k file projects
* **Startup Time**: <20 seconds (CodeGen-2B only)

### **Quality Targets**

* **Accuracy**: 95% relevant responses
* **User Satisfaction**: >4.5/5 rating
* **Template Coverage**: 50+ templates covering all common patterns
* **Fallback Success**: 100% handling of edge cases

This hybrid architecture delivers enterprise-grade code analysis with minimal resource usage, instant responses for common queries, and intelligent handling of complex requests.