Prompt Engine

Documentation

A Custom Trainable NLP Engine for
Natural Language to React Components

Version 1.0.0

Table of Contents

1. Overview

The Prompt Engine is a custom Natural Language Processing (NLP) system designed specifically for converting text prompts into React components. Unlike generic AI solutions, this engine is built to understand YOUR UI kit vocabulary and can be trained to improve over time.

Key Features:

• Custom NLP pipeline optimized for UI component generation

• Trainable classifiers that learn from corrections

• No external API required - works completely offline

• Plugin system for optional AI integration (Claude, OpenAI)

• Confidence scoring to know when output is uncertain

• Schema-driven - define your own component vocabulary

What It Can Do:

• "create a primary button" →

• "large danger button with icon" →

• "three cards in a row" → Grid with 3 Card components

• "login form with email and password" → Form with Input components

• "modal containing a form" → Nested Modal > Form structure

2. Architecture

The engine follows a pipeline architecture where each stage processes and enriches the understanding of the user's prompt.

Processing Pipeline:

User Prompt → **Lexer** (tokenization) → **Semantic Analyzer** (understanding) → **Intent Resolver** (what to do) → **Entity Extractor** (components, props) → **Context Builder** (unified context) → **JSX Generator** (output code)

Stage	Purpose	Output
Lexer	Break prompt into tokens	Tokens, phrases
Semantic Analyzer	Understand grammar & meaning	Roles, relationships
Intent Resolver	Classify user intent	Intent type + confidence
Entity Extractor	Find components & modifiers	Entities list
Context Builder	Combine all analysis	Processing context
JSX Generator	Generate React code	JSX + imports

3. Installation & Setup

Prerequisites:

• Node.js 16 or higher

• npm or yarn

• TypeScript 5.x

Installation:

```
npm install
```

Build:

```
npm run build
```

Run Examples:

```
npm run example
```

Development Mode:

```
npm run dev
```

4. File Structure

```
prompt-engine/
■■■ src/
■ ■■■ core/ # Processing pipeline
■ ■ ■■■ Engine.ts # Main orchestrator
■ ■ ■■■ Lexer.ts # Tokenization
■ ■ ■■■ SemanticAnalyzer.ts # NLP understanding
■ ■ ■■■ IntentResolver.ts # Intent classification
■ ■ ■■■ EntityExtractor.ts # Entity extraction
■ ■ ■■■ ContextBuilder.ts # Context assembly
■ ■
■ ■■■ knowledge/ # Knowledge base
■ ■ ■■■ ComponentGraph.ts # UI kit graph
■ ■
■ ■■■ generators/ # Code generation
■ ■ ■■■ JSXGenerator.ts # React JSX output
■ ■
■ ■■■ plugins/ # Plugin system
■ ■ ■■■ PluginManager.ts # Plugin orchestration
■ ■ ■■■ LocalAIPlugin.ts # Default local plugin
■ ■ ■■■ ClaudePlugin.ts # Anthropic Claude
■ ■ ■■■ OpenAIPlugin.ts # OpenAI GPT
■ ■
■ ■■■ types/ # TypeScript definitions
■ ■ ■■■ index.ts
■ ■
■ ■■■ usePromptEngine.ts # React hook
■ ■■■ index.ts # Main exports
■
■■■ examples/
■ ■■■ schema.ts # Example UI kit schema
■ ■■■ training.ts # Training examples
■
■■■ docs/
■ ■■■ documentation.pdf # This document
■
■■■ package.json
■■■ tsconfig.json
■■■ README.md
```

5. Core Components

5.1 Engine.ts

The main orchestrator that coordinates all processing stages. It initializes the pipeline, manages plugins, and exposes the public API.

Key Methods:

initialize(schema): Load UI kit schema and initialize all processors

process(prompt): Process a prompt and return generated JSX

learn(prompt, correction, expected): Train from a correction

exportTrainingData(): Export training examples for persistence

importTrainingData(data): Import previously saved training

5.2 Lexer.ts

Handles tokenization - breaking the input text into meaningful tokens. Recognizes multi-word phrases (e.g., "two column", "text field") and normalizes text for consistent processing.

Features:

• Multi-word phrase recognition

• Stop word filtering

• Basic stemming

• Custom phrase addition

5.3 SemanticAnalyzer.ts

The brain of the system. Performs grammatical analysis (nouns, verbs, adjectives), extracts semantic roles (action, target, modifiers), and maps understanding to your UI kit domain.

What It Extracts:

Grammar: Nouns (components), verbs (actions), adjectives (modifiers)

Semantic Roles: Action (create), target (button), modifiers (primary, large)

Relationships: Containment (button inside card), siblings (button and input)

Domain Mapping: Maps generic words to your specific components

5.4 IntentResolver.ts

Classifies the user's intent using a trainable Naive Bayes classifier. Determines what the user wants to do: create a component, build a layout, combine components, etc.

Intent Types:

create_component: Create a single component (button, input, card)

create_layout: Create a layout structure (grid, columns, flex)

create_page: Create a full page (landing, dashboard, login)

combine: Combine multiple components (form with inputs)

modify: Modify existing component (change size, add prop)

query: Ask about available components

Training: The classifier learns from corrections. When you call engine.learn(), it adds examples to improve future classification.

5.5 EntityExtractor.ts

Extracts specific entities from the analyzed prompt: components, modifiers (variant, size, state), quantities, layout configurations, and additional props.

Entity Types:

component: UI components (button, card, input)

modifier: Variants (primary), sizes (lg), states (disabled)

quantity: Numbers (3 cards, two columns)

layout: Layout configurations (grid, flex direction)

prop: Additional properties (with icon, rounded)

5.6 ContextBuilder.ts

Combines all analysis results into a unified ProcessingContext object. Calculates coverage (how much we understood), validates completeness, and provides helper methods for generators.

5.7 ComponentGraph.ts

The knowledge base that stores your UI kit as a searchable graph. Understands component relationships, synonyms, and provides fuzzy matching for user input.

Capabilities:

• Synonym resolution (btn → button, cta → button)

• Fuzzy matching for typos and variations

• Category-based organization

• Relationship tracking (what can contain what)

• Training data generation from schema

5.8 JSXGenerator.ts

Produces React JSX code from the ProcessingContext. Handles single components, nested structures, layouts, and page templates.

Generation Modes:

Single Component:

Nested Structure:

Layout: ...

Page Template: Full page with sections from template

6. Plugin System

The plugin system allows extending the engine with external AI services or custom processing logic. Plugins can enhance context understanding when local confidence is low.

Built-in Plugins:

LocalAIPlugin: Default plugin using local heuristics

ClaudePlugin: Integration with Anthropic Claude API

OpenAIPlugin: Integration with OpenAI GPT API

Using Plugins:

```
// Register a plugin
const claude = new ClaudePlugin({ apiKey: 'your-key' });
await engine.plugins.register('claude', claude);

// Set as active
engine.plugins.setActive('claude');

// Plugin is automatically used when confidence is low
```

Creating Custom Plugins:

Implement the Plugin interface with initialize(), enhance(), and optionally generate() methods.

7. Training the Engine

The engine improves through training. When it produces incorrect output, you can provide corrections that it learns from.

Basic Training:

```
await engine.learn(
'make a CTA', // Original prompt
'CTA means call-to-action', // Description
{ // Expected output
jsx: 'Get Started',
imports: ["import { Button } from '@/components/ui';"]
}
);
```

Batch Training:

Import multiple training examples at once using importTrainingData(). This is useful for loading previously saved training or pre-training with a dataset.

Training Tips:

• Train with common variations of the same intent

• Include both formal and informal phrasings

• Add domain-specific terminology your users might use

• Export and persist training data regularly

• Test trained prompts to verify improvement

See examples/training.ts for comprehensive examples.

8. Usage Examples

Basic Usage:

```
import { PromptEngine } from 'prompt-engine';
import schema from './my-schema';

const engine = new PromptEngine();
await engine.initialize(schema);

const result = await engine.process('create a primary button');
console.log(result.jsx);
//
```

React Hook:

```
import { usePromptEngine } from 'prompt-engine';

function MyComponent() {
const { ready, generate } = usePromptEngine(schema);

const handleGenerate = async () => {
const result = await generate('large danger button');
console.log(result.jsx);
};

if (!ready) return Loading...;
return Generate;
}
```

With Debug Info:

```
const engine = new PromptEngine({ debug: true });
await engine.initialize(schema);

const result = await engine.process('card with button');

console.log('Intent:', result.debug.intent);
console.log('Entities:', result.debug.entities);
console.log('Confidence:', result.confidence);
```

9. API Reference

PromptEngine

constructor(config?): Create engine instance

initialize(schema): Initialize with UI kit schema

process(prompt): Process prompt, returns EngineResult

learn(prompt, correction, expected): Train from correction

getComponents(): Get list of component names

getComponent(name): Get component details

exportTrainingData(): Export training examples

importTrainingData(data): Import training examples

reset(): Reset to initial state

isReady: Check if initialized

plugins: Access PluginManager

EngineResult

jsx: string - Generated JSX code

imports: string[] - Required import statements

confidence: number - Overall confidence (0-1)

processingTime: number - Time in milliseconds

error: string? - Error message if failed

suggestions: string[]? - Suggestions if uncertain

debug: object - Debug information (tokens, intent, entities)

UIKitSchema

name: string - UI kit name

version: string - Version number

components: Record - Component definitions

layouts: Record? - Layout templates

pages: Record? - Page templates