

Dify

# 超越界面：前端工程师如何塑造 AI 原生应用的未来

Beyond UI: How Frontend Engineers Can Shape the Future of AI-Native  
Applications

# About Me



# Frontend Software Engineer @ Workflow & RAG Squad



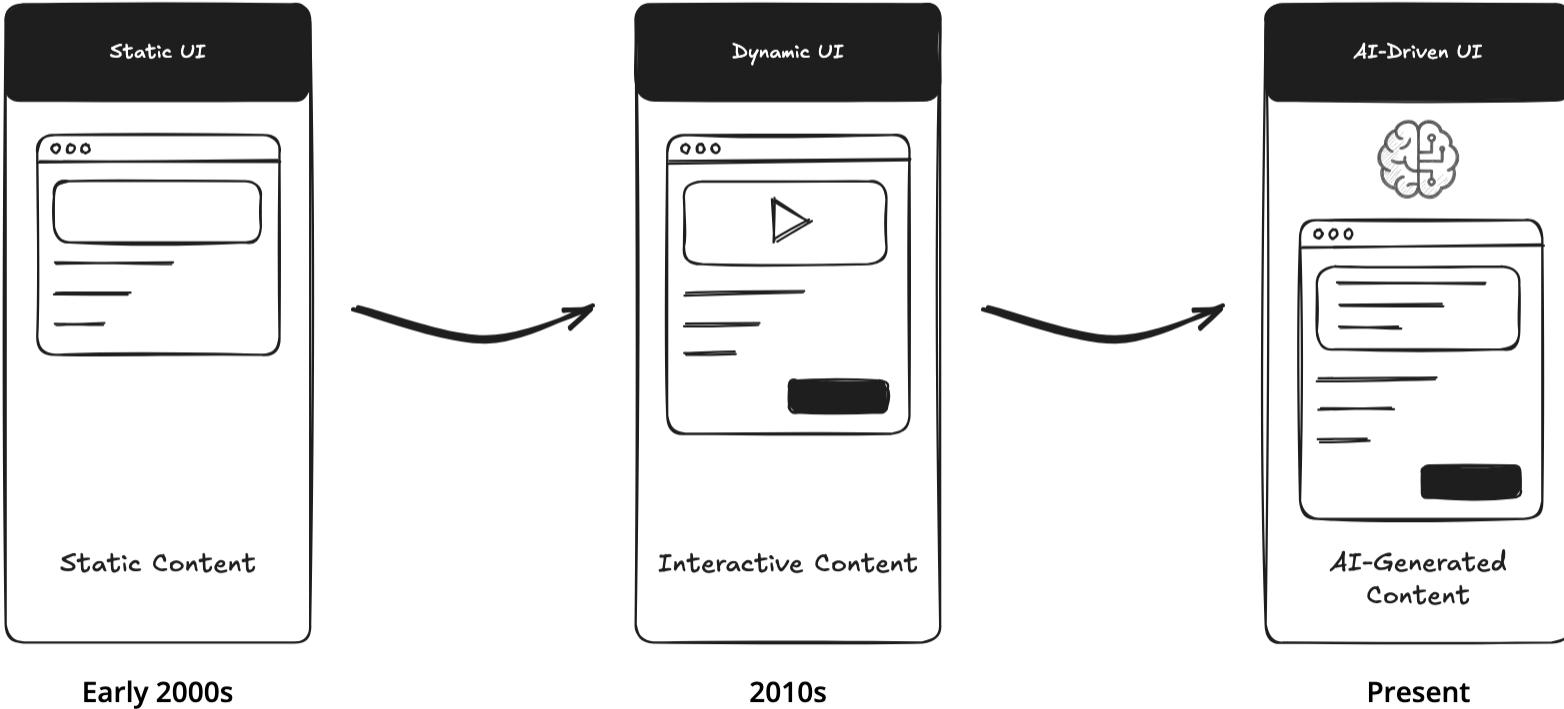
# Outline

30 min · 4 parts | 5 min · Q&A

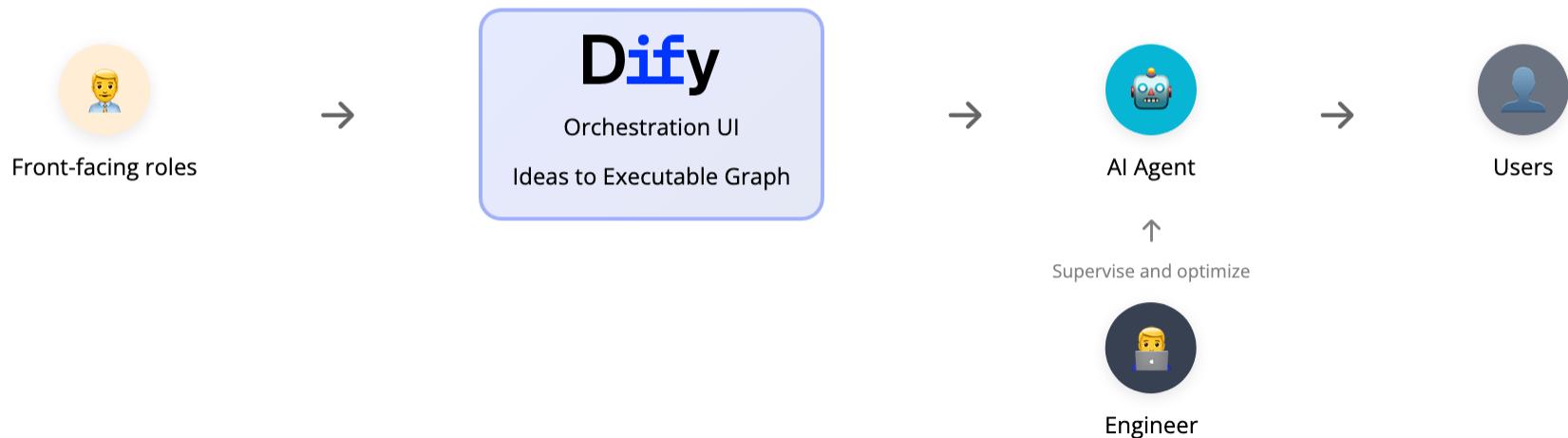
- **Part 1 — The Evolution of Frontend Development**  
From deterministic to probabilistic
- **Part 2 — Dify**  
Why **workflow** is a good choice?
- **Part 3 — Frontend Design & Architecture Patterns**  
All for **extensibility**
- **Part 4 — Live Demo**  
Use **Workflow & Knowledge Pipeline** to build an AI application in a few minutes

# The Evolution of Frontend Development

## From deterministic to probabilistic



# From Expert to Everyone



# Agentic Workflow

Drag and drop, orchestrate the nodes, and turn your AI vision into reality.

The screenshot shows the Dify Studio interface, specifically the Chatbot Playground section. The top navigation bar includes 'EXPLORE', 'STUDIO / CHATBOT PLAYGROUND', 'KNOWLEDGE', 'TOOLS', 'PLUGINS', and a 'Test run' button with a '2' notification. On the left, there's a sidebar with icons for Auto-Saved (21:02:35), Unpublished, and various project management tools. The main workspace displays a workflow diagram:

```
graph LR; DE[DOC EXTRACTOR] --> LLM[LLM<br/>GPT-4o]; LLM --> CODE[CODE<br/>ON FAILURE: Fail Branch];
```

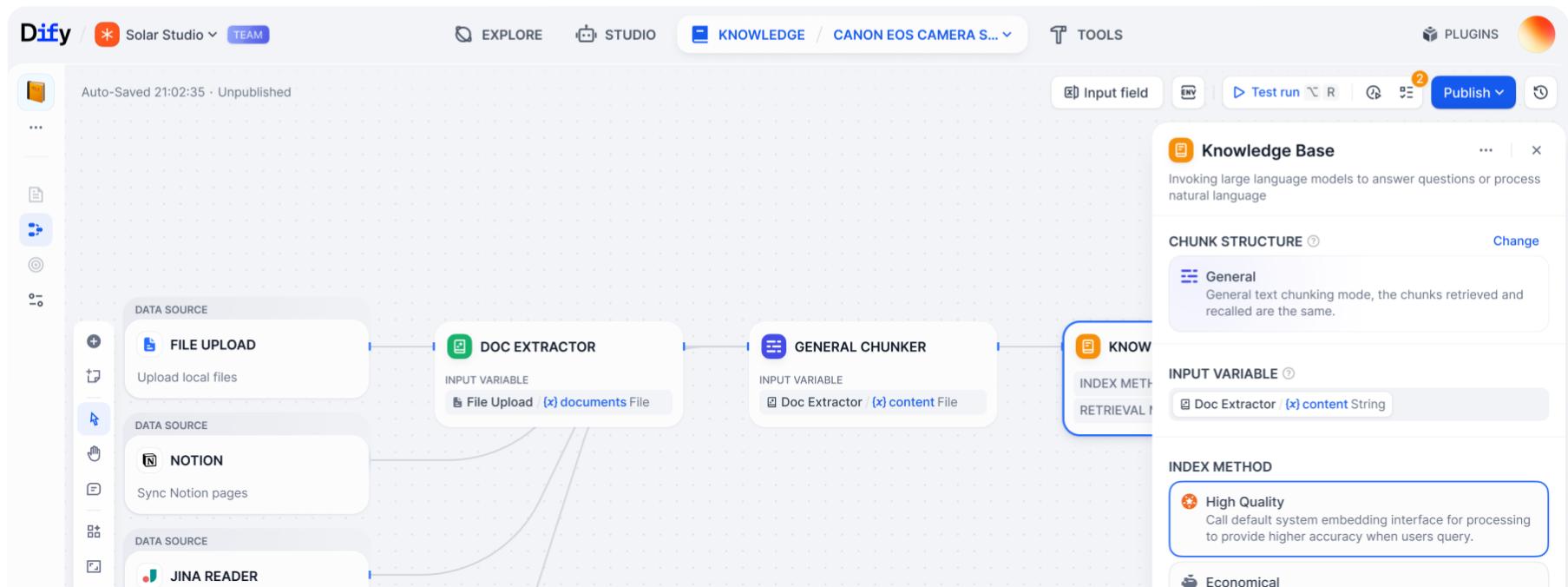
The 'DOC EXTRACTOR' node has an input variable 't {x}sys.files File'. The 'LLM' node is detailed as "Invoking large language models to answer questions or process natural language" with a 'GPT-4o' model selected. The 'CODE' node has an 'ON FAILURE' branch labeled 'Fail Branch'. The 'Fail Branch' leads to a 'GRO' (Group) node, which then connects to a 'PROMPT' node. The 'PROMPT' node contains a Jinja template with variables and a note about handling errors. The 'USER' node at the bottom provides instructions for entity extraction. To the right, the 'Test Run#5' panel shows the execution flow and results:

Step	Action	Time	Status
1	START	0.02 s	✓
2	DOC EXTRACTOR	0.27 s	✓
3	LLM	258 tokens - 0.12 s	✓
4	CODE	1.53 s	⚠ Node exception, will automatically execute the fail branch. The node output will return an error type and error message and pass them to downstream.

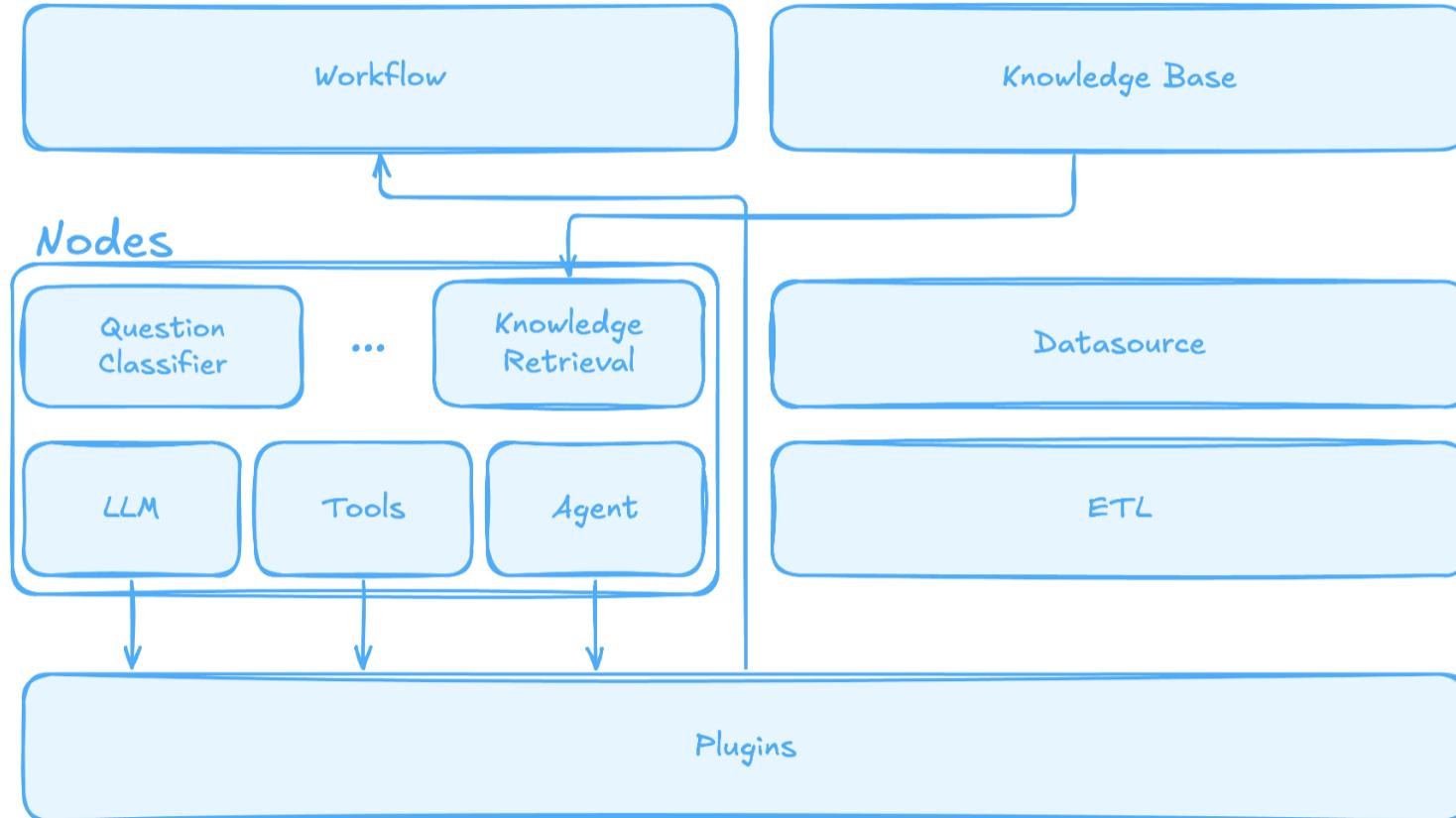
The 'INPUT' tab shows the raw JSON input for the 'CODE' step. The 'OUTPUT' tab shows the resulting JSON output.

# Knowledge Pipeline

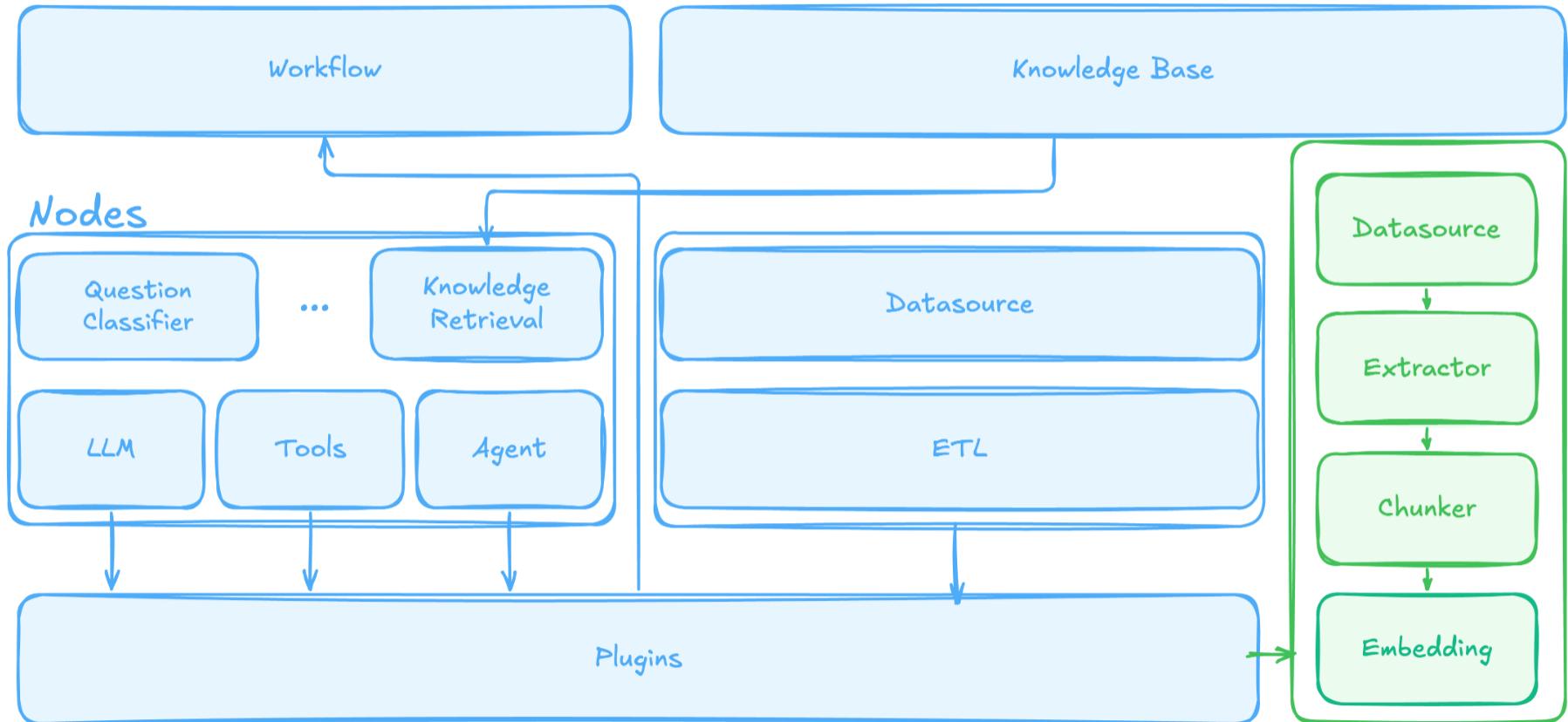
Visually customize your data processing flow for better accuracy and relevance.



# Original Workflow & RAG System



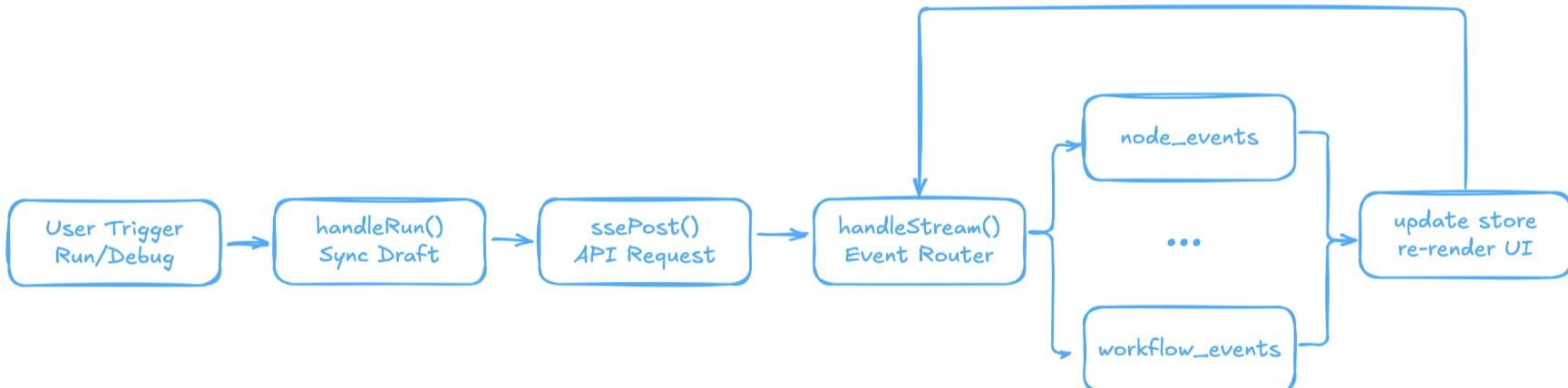
# New Workflow & RAG System



# Visual Workflow Builder

Features	Implementation
Canvas	ReactFlow with custom node/edge types
State Management	Zustand with sliced stores
Real-time	SSE streaming for execution events
Debugging	Built-in debug panel for tracing execution
History	Version control and draft management

# Execution Lifecycle



**One Workflow Builder**  
**Multiple Applications**

# Store Slice Injection

```
export type SliceFromInjection
= Partial<WorkflowAppSliceShape>
& Partial<RagPipelineSliceShape>

export const createWorkflowStore = (params: CreateWorkflowStoreParams) => {
  const { injectWorkflowStoreSliceFn } = params || {}

  return createStore<Shape>((...args) => ({
    ...createChatVariableSlice(...args),
    ...createEnvVariableSlice(...args),
    ...createNodeSlice(...args),
    ...createWorkflowSlice(...args),
    // ... more base slices
    ...(injectWorkflowStoreSliceFn?.(...args) || {} as SliceFromInjection),
  }))
}
```

Core store stays **stable**

Apps inject only **what they own**

# Store Slice Injection

```
export type SliceFromInjection
= Partial<WorkflowAppSliceShape>
& Partial<RagPipelineSliceShape>

export const createWorkflowStore = (params: CreateWorkflowStoreParams) => {
  const { injectWorkflowStoreSliceFn } = params || {}

  return createStore<Shape>((...args) => ({
    ...createChatVariableSlice(...args),
    ...createEnvVariableSlice(...args),
    ...createNodeSlice(...args),
    ...createWorkflowSlice(...args),
    // ... more base slices
    ...(injectWorkflowStoreSliceFn?.(...args) || {} as SliceFromInjection),
  }))
}
```

Core store stays **stable**

Apps inject only **what they own**

# Workflow

```
export type WorkflowSliceShape = {
  appId: string
  appName: string
  notInitialWorkflow: boolean
  showOnboarding: boolean
  nodesDefaultConfigs: Record<string, any>
  // ... workflow-app specific state
}

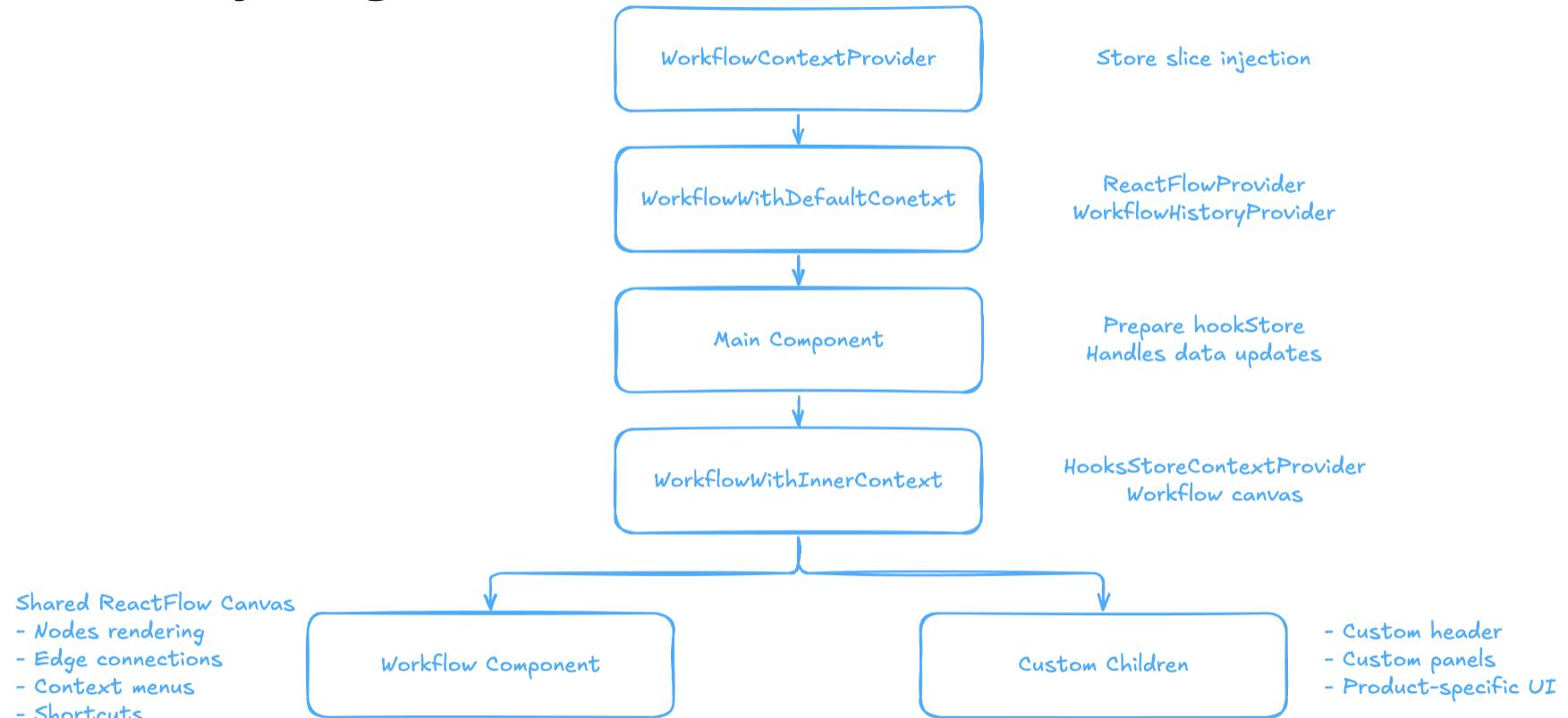
const WorkflowAppWrapper = () => {
  return (
    <WorkflowContextProvider
      injectWorkflowStoreSliceFn={createWorkflowSlice}
    >
      <WorkflowAppWithAdditionalContext />
    </WorkflowContextProvider>
  )
}
```

# Knowledge Pipeline

```
export type RagPipelineSliceShape = {
  pipelineId: string
  knowledgeName: string
  knowledgeIcon?: IconInfo
  showInputFieldPanel: boolean
  ragPipelineVariables: RAGPipelineVariables
  dataSourceList: ToolWithProvider[]
  // ... RAG-specific state
}

const RagPipelineWrapper = () => {
  return (
    <WorkflowContextProvider
      injectWorkflowStoreSliceFn={createRagPipelineSliceS
    >
      <RagPipeline />
    </WorkflowContextProvider>
  )
}
```

# Hierarchy Diagram



# Basic Types

- string
- number
- boolean
- file
- array
  - array[string]
  - array[number]
  - array[boolean]
  - array[file]
  - array[object]
- object

# Special Types



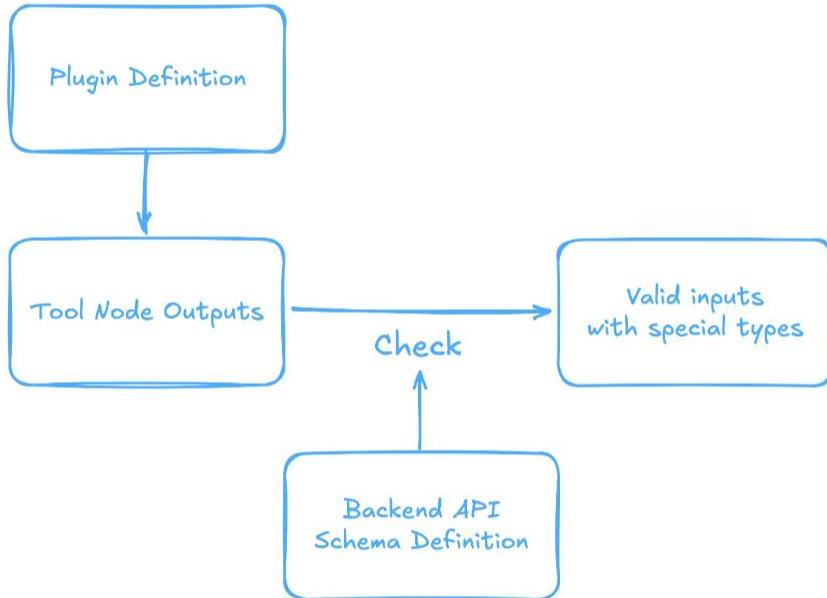
General Chunker

array[string]

general\_structure

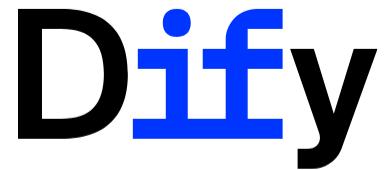
array[string]

# Solution



The screenshot shows a software interface for managing a 'Knowledge Base'. At the top, there's a header with a 'Knowledge Base' icon, window controls, and tabs for 'SETTINGS' (which is selected) and 'LAST RUN'. Below the tabs, there's a section titled 'CHUNK STRUCTURE' with a 'General' sub-section. This section describes how the knowledge base splits content into chunks and retrieves the most relevant ones based on user queries. It mentions that users can customize chunking rules like delimiter, maximum length, and overlap. It also notes that preprocessing options are available to clean up text by removing excess spaces, URLs, and smilies. There are two cards below this section: one for 'GENERAL CHUNKER' and one for 'KNOWLEDGE BASE'. The 'KNOWLEDGE BASE' card shows the 'INDEX METHOD' is set to 'Economical' and the 'RETRIEVAL SETTING' is 'Inverted index'. In the 'CHUNKS' section, a message indicates that the general chunker is selected. The 'INDEX METHOD' section contains two options: 'High Quality' (selected) and 'Economical'. The 'High Quality' section explains that calling the embedding model for processing documents helps generate high-quality answers. The 'Economical' section notes that using 10 keywords per chunk for retrieval consumes no tokens but results in reduced retrieval accuracy. At the bottom, there's a slider for 'Number of Keywords' set to 10, and a progress bar indicating 60% completion.

# Live Demo



# Thank you

**WEBSITE**  
[dify.ai](https://dify.ai)

**GITHUB**  
[github.com/langgenius/dify](https://github.com/langgenius/dify)

**JOIN US**  
[joinus@dify.ai](mailto:joinus@dify.ai)