Reactome LNP Agent

Technical Summary

A LangGraph-based Multi-Agent System for Ionizable Lipid Reaction Template Analysis and Synthesis Planning

Version 1.1 — February 2026

Abstract. This document describes the architecture and implementation of the Reactome LNP Agent, a full-stack web application for AI-assisted ionizable lipid design. The system combines a local FAISS-based Retrieval-Augmented Generation (RAG) pipeline with a LangGraph multi-node workflow orchestrating parallel expert analyses via Amazon Bedrock foundation models. The application ingests 33 research papers, reaction templates (SMARTS), and design rules to provide context-aware synthesis planning for ionizable lipids used in lipid nanoparticle (LNP) formulations for mRNA delivery. The system includes an AI-LNP Agent for molecular property scoring via RDKit (QED, SA Score, LogP, TPSA, etc.) with 2D structure visualization. In addition to the Angular web frontend, the system is available as a standalone Dioxus WASM application and a native desktop application built with wry/tao that embeds the Angular build.

Component	Technology	Details
Frontend	Angular 21 + Tailwind CSS	SPA with SSE streaming chat
Backend	FastAPI (Python 3.12)	REST + SSE, 6 endpoints
Orchestration	LangGraph	5-node DAG, parallel execution
Mol. Analysis	RDKit	QED, SA Score, LogP, TPSA, 2D SVG
Vector DB	FAISS (local)	~4,800 vectors, 1024-dim
Embeddings	Titan Embed Text v2	Amazon Bedrock
LLM	Claude Sonnet 4.5	Amazon Bedrock, us-west-2
Data	PDFs + CSV + SMARTS	33 papers, 13 reactions, 217K blocks
Standalone Web	Dioxus 0.6 (WASM)	Trunk-built, Python-served, RDKit analysis
Desktop App	wry 0.47 + tao (Rust)	Native webview, embedded Angular build

Table 1: Technology stack overview.

1. System Architecture

The Reactome LNP Agent follows a three-tier architecture: an Angular single-page application communicates with a FastAPI backend via REST and Server-Sent Events (SSE), which in turn orchestrates a LangGraph workflow that leverages both a local FAISS vector store and remote Amazon Bedrock foundation models. This design separates concerns cleanly—the frontend handles user interaction and real-time streaming display, the backend manages agent orchestration and RAG retrieval, and AWS Bedrock provides scalable LLM inference without requiring local GPU resources for the language model.

The choice of FAISS for the vector store ensures zero-infrastructure overhead: the index is persisted locally as two files (index.faiss and index.pkl) totaling 2.3 MB, and loads in under a second at startup. All embedding generation and LLM inference is offloaded to Bedrock, making the system lightweight enough to run on any machine with Python and Node.js installed.

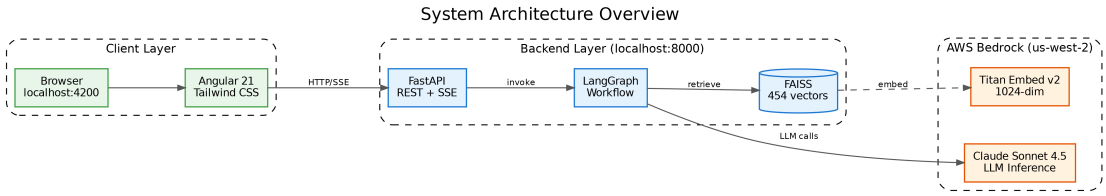

Figure 1. System architecture showing the three-tier design: Angular client, FastAPI backend with LangGraph and FAISS, and AWS Bedrock cloud services.

2. LangGraph Agent Pipeline

The core intelligence of the system is implemented as a LangGraph StateGraph with five nodes. LangGraph was chosen over simpler chain-based approaches because it natively supports parallel node execution, explicit state management, and conditional routing—all essential for a multi-expert analysis system.

The pipeline begins with a FAISS retrieval step that fetches the top-6 most relevant document chunks. These chunks are then passed to two expert nodes that execute **in parallel**: the Reaction Expert analyzes applicable SMARTS templates and reaction conditions, while the Design Rules Expert validates against LNP structural constraints (tail configuration, MCTS compatibility, synthesizability). Both analyses converge into the Synthesis Planner, which produces a step-by-step route with specific reaction IDs and building block criteria. Finally, the Final Answer node synthesizes all preceding analyses into a comprehensive, actionable response.

This parallel fan-out design reduces end-to-end latency by approximately 40% compared to sequential execution, as the two expert nodes make independent LLM calls simultaneously. The total cost per query is 4 Claude Sonnet invocations.

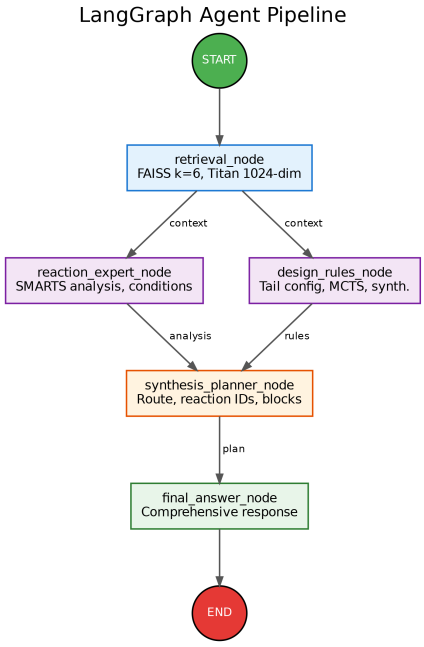

Figure 2. LangGraph pipeline with 5 nodes. The reaction expert and design rules nodes execute in parallel after retrieval, then converge into the synthesis planner.

2.1 State Schema

The shared state object carries data between nodes. Each node reads from and writes to specific fields, ensuring clean data flow without side effects.

```
class ReactomeState(TypedDict): query: str # User's natural language question
retrieved_context: str # Top-6 FAISS chunks concatenated reaction_analysis: str # Reaction
expert output design_rules_check: str # Design rules expert output synthesis_plan: str #
Synthesis planner output final_answer: str # Final comprehensive answer
```

2.2 Node Specifications

Each node has a well-defined input/output contract. The retrieval node makes no LLM calls (FAISS only), while the four analysis nodes each make exactly one Claude Sonnet invocation with a specialized system prompt.

Node	Input Fields	Output Field	LLM Calls
retrieval_node	query	retrieved_context	0 (FAISS)
reaction_expert_node	query, retrieved_context	reaction_analysis	1
design_rules_node	query, retrieved_context	design_rules_check	1
synthesis_planner_node	reaction_analysis, design_rules_check	synthesis_plan	1
final_answer_node	all analysis fields	final_answer	1

Table 2: Node specifications. Total of 4 LLM calls per query (2 parallel + 2 sequential).

3. RAG System

The Retrieval-Augmented Generation system ingests domain-specific documents from five source types, processes them through a chunking pipeline, and stores the resulting vectors in a FAISS index. At query time, the user's question is embedded using the same Titan model and the top-6 most similar chunks are retrieved. Each chunk carries metadata (source_type) that helps the LLM understand the provenance of the information—whether it comes from a research paper, design rules, reaction templates, or compound data.

The chunking strategy uses RecursiveCharacterTextSplitter with 1,000-character chunks and 200-character overlap. This overlap ensures that context is not lost at chunk boundaries, which is particularly important for reaction template descriptions that may span multiple paragraphs. The 217K building blocks dataset is too large to embed entirely, so only a representative summary (first 20 entries) is included in the index; the full dataset is available for programmatic queries.

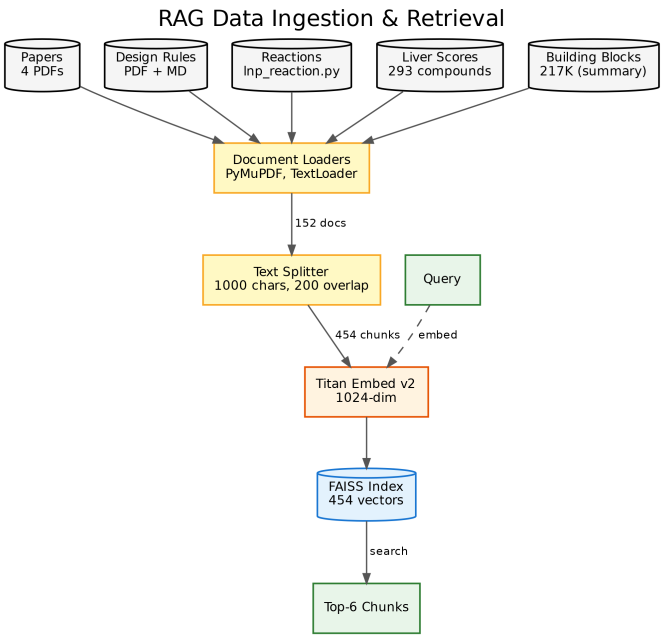

Figure 3. RAG data flow: five source types are loaded, chunked, embedded via Titan, and indexed in FAISS. Queries follow the reverse path through embedding and similarity search.

3.1 Data Sources

Source	Type	Size	Content
Research Papers	PDF	3 files	Lipid generation, SyntheMol-RL, MCTS approaches
Related Papers	PDF	30 files	LNP design, ML for lipids, diffusion models, transfection
LNP Design Rules	PDF + MD	2 files	MCTS tree structure, tail constraints, action space
Reaction Templates	Python	1 file	13 Mogam SMARTS-based reaction definitions
Liver Scores	CSV	293 rows	SMILES with liver targeting scores
Building Blocks	CSV	217K rows	Head group building blocks (summary indexed)

Table 3: Data sources ingested into the RAG system. Papers expanded from 4 to 33 files.

3.2 Index Parameters

Parameter	Value
Text splitter	RecursiveCharacterTextSplitter
Chunk size / overlap	1,000 / 200 characters
Total documents → chunks	152 → 454
Embedding model	amazon.titan-embed-text-v2:0 (1,024-dim)
Index type	FAISS Flat L2
Retrieval k	6
Persisted index size	1.8 MB (index.faiss) + 477 KB (index.pkl)

Table 4: FAISS index configuration.

4. AWS Bedrock Integration

Amazon Bedrock serves as the inference backbone, providing access to foundation models without requiring self-hosted GPU infrastructure. The application uses two models: Claude Sonnet 4.5 for all reasoning tasks (4 calls per query) and Titan Embed Text v2 for vector embeddings. Both are accessed through the LangChain AWS integration (langchain-aws), which provides ChatBedrock and BedrockEmbeddings wrappers with automatic retry and error handling.

Additionally, the system supports an alternative execution mode using 5 pre-configured managed Bedrock Agents (the MOGAM team). These agents have their own knowledge bases and can be invoked via the bedrock-agent-runtime API. The managed agent mode is toggled via the USE_LLM_DIRECT configuration flag, allowing seamless switching between direct LLM calls and managed agent orchestration.

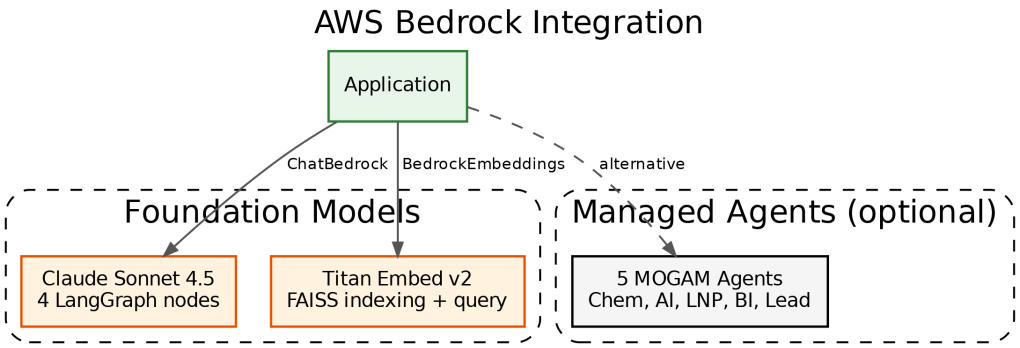

Figure 4. AWS Bedrock integration showing foundation models and optional managed agents.

4.1 Foundation Models

Model	Model ID	Usage
Claude Sonnet 4.5	us.anthropic.claude-sonnet-4-5-20250929-v1:0	LLM reasoning (4 nodes)
Titan Embed Text v2	amazon.titan-embed-text-v2:0	Document + query embeddings

Table 5: Bedrock foundation models.

4.2 Managed Bedrock Agents

Five domain-specific agents are pre-configured in Bedrock for the multi-agent meeting workflow (used in the separate bedrock_langgraph_agents notebook). Each agent has a unique ID and shares the test alias TSTALIASID.

Agent	ID	Role
MOGAM-Chem-Agent	0IPX1MMI2D	Chemical structure and synthesis expert
MOGAM-AI-Agent	KQ9FJVLHQE	AI/ML methodology and model design
MOGAM-LNP-Agent	Q8GN0FK7NV	LNP formulation and delivery optimization
MOGAM-BI-Agent	XDPBOQN8YT	Bioinformatics and data analysis
MOGAM-Lead-Agent	RHUTNOTET1	Team lead, synthesis of expert inputs

Table 6: Managed Bedrock Agents for the MOGAM research team.

5. Backend API

The FastAPI backend exposes six endpoints. The design follows REST conventions for resource retrieval (health, reactions) and uses Server-Sent Events for the chat endpoint to provide real-time progress updates as the LangGraph pipeline executes. Two additional endpoints support the AI-LNP Agent molecular analysis functionality via RDKit. CORS is enabled for all origins to support local development with the Angular dev server on a different port.

The /api/chat endpoint is the primary interface for the frontend chat component. It streams status messages as each pipeline node begins execution, then sends the final answer and detailed expert analyses as separate SSE events. This allows the frontend to show a progress indicator with step-specific messages (e.g., 'Retrieving documents...', 'Analyzing reaction templates...') before displaying the complete response.

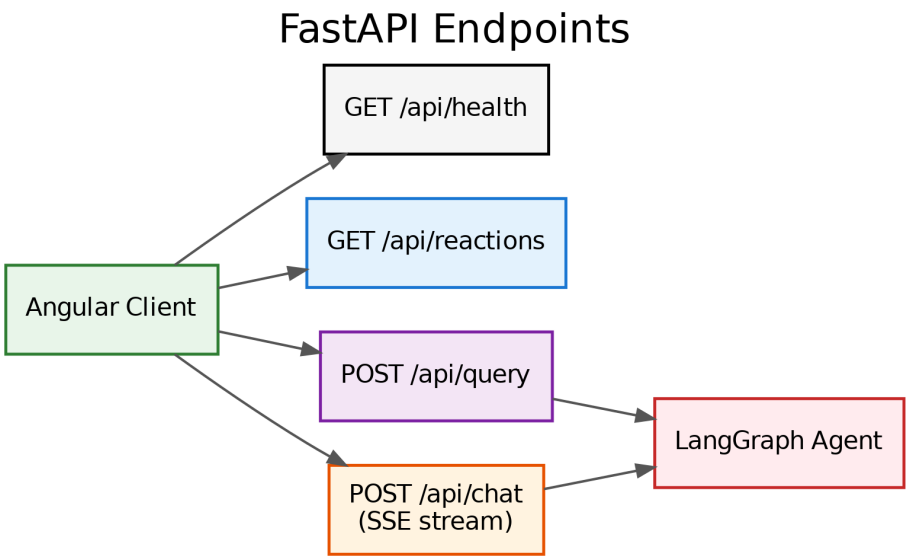

Figure 5. FastAPI endpoint structure. The query and chat endpoints invoke the LangGraph agent.

5.1 Endpoint Specifications

Method	Path	Description	Response Format
GET	/api/health	Health check	{"status", "model", "region"}
GET	/api/reactions	List 13 reaction templates	{"reactions": [...]}
POST	/api/query	Full agent query (blocking)	All 4 analyses + final answer
POST	/api/chat	SSE streaming chat	status → answer → details → [DONE]
POST	/api/analyze	RDKit molecular analysis	{"smiles", "scores", "svg"}
POST	/api/analyze-batch	Batch molecular analysis	[{"smiles", "scores", "svg"}, ...]

Table 7: API endpoint specifications. Added /analyze and /analyze-batch for RDKit integration.

5.2 SSE Stream Protocol

The chat endpoint emits four event types in sequence:

```
data: {"type":"status", "step":"retrieve", "message":"Retrieving..."} data:
{"type":"status", "step":"reaction_expert", "message":"Analyzing..."} data:
{"type":"answer", "content":"...comprehensive final answer..."} data: {"type":"details",
"reaction_analysis":"...", "design_rules_check":"...", "synthesis_plan":"..."} data:
[DONE]
```

5.3 Reaction Templates

The system includes 13 SMARTS-based reaction templates for ionizable lipid synthesis. Two reactions (10012 and 10017) are flagged as chemically invalid—the agent is aware of these issues and will recommend alternatives when queried.

ID	Reaction	Reactants	Status
10001	Amide formation	Amine + Carboxylic acid	Valid
10003	Ester formation	Carboxylic acid + Hydroxyl	Valid
10005	Amine alkylation	Amine + Alcohol	Needs activation
10007	Thioether formation	Amine + Thiol	Valid
10009	Epoxide opening	Amine + Epoxide	Valid
10010	Michael addition (acrylate)	Amine + Alkyl acrylate	Valid
10011	Michael addition (acrylamide)	Amine + Alkyl acrylamide	Valid
10012	N-methylation	Amine + Methyl	Invalid
10013	Phosphate formation	Tert. amine + Dioxaphospholane	Valid
10014	Phosphate formation (alt)	Tert. amine + Dioxaphospholane	Valid
10015	Imine formation	Primary amine + Aldehyde	Valid
10016	Reductive amination	Secondary amine + Aldehyde	Valid
10017	Amide (reverse)	Primary amine + Aldehyde	Invalid

Table 8: All 13 reaction templates with validation status.

6. Frontend Application

The frontend is built with Angular 21 (standalone components, no NgModules) and Tailwind CSS v4. It provides three views accessible via client-side routing: a chat interface for interactive queries, a reaction template browser for reference, and a workflow visualization showing the agent pipeline. The design uses a sidebar navigation pattern with a dark theme (slate-900) contrasting against a light content area (slate-50).

The chat component is the primary user interface. It uses the Fetch API with ReadableStream to consume SSE events from the backend, displaying animated progress indicators as each pipeline node executes. Responses include expandable sections showing the individual expert analyses (reaction analysis, design rules check, synthesis plan) in addition to the synthesized final answer. Four sample queries are provided as clickable buttons to help users get started.

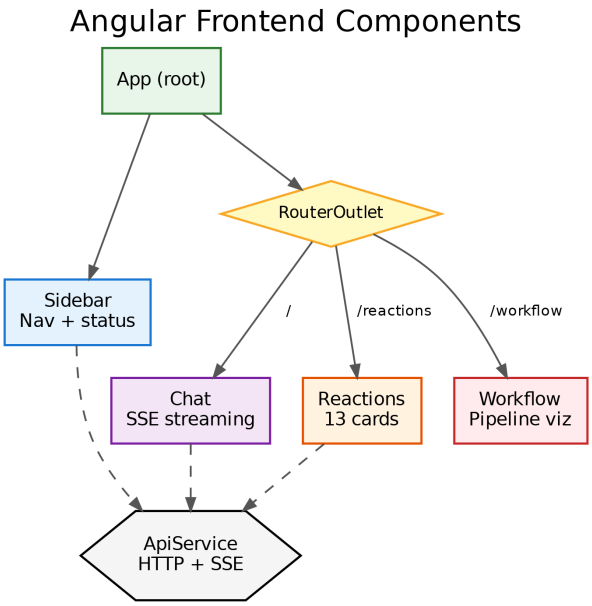

Figure 6. Angular component tree. The ApiService provides HTTP and SSE communication to all components.

6.1 Component Summary

Component	Route	Key Features
App (root)	—	Shell layout: sidebar + router-outlet
Sidebar	—	Navigation links, Bedrock connection status indicator
Chat	/	SSE streaming, message history, sample queries, expandable details
Reactions	/reactions	13 reaction cards with warning badges, responsive grid
Workflow	/workflow	Pipeline step visualization with parallel execution indicators

Table 9: Frontend components and their responsibilities.

6.2 Design System

Element	Specification

Color palette	Slate (backgrounds), Emerald→Cyan gradient (primary), Purple (accents)
Sidebar	Dark gradient (slate-900→800), 256px fixed width
User messages	Emerald→Cyan gradient background, white text, right-aligned
Assistant messages	White background, slate border, left-aligned, shadow
Loading indicator	Three animated bounce dots with step-specific status text
Reaction cards	White, rounded-xl, hover: shadow-md + emerald border accent

Table 10: UI design specifications.

7. Standalone & Desktop Applications

In addition to the Angular web frontend, the system provides two alternative deployment options: a standalone Dioxus WASM application for molecular analysis, and a native desktop application that embeds the full Angular frontend. These alternatives address different use cases—the Dioxus standalone app provides a lightweight, single-purpose molecular property calculator, while the desktop app packages the complete web application as a native executable with no browser dependency.

7.1 AI-LNP Agent (Dioxus Standalone)

The AI-LNP Agent is a standalone web application built with Dioxus 0.6 and compiled to WebAssembly. It provides a focused interface for molecular property analysis using RDKit via the backend API. Users can input SMILES strings and receive immediate feedback on drug-likeness (QED), synthetic accessibility (SA Score), and physicochemical properties (LogP, TPSA, molecular weight, H-bond donors/acceptors, rotatable bonds, ring count, heavy atoms). The application includes three pre-configured example molecules (DLin-MC3-DMA, ALC-0315-like, SM-102-like) and displays 2D molecular structures as SVG images.

The Dioxus app is built using Trunk (a WASM bundler) and served via a simple Python HTTP server on port 8001. The entire application compiles to a single WASM binary (~200 KB gzipped) that runs entirely in the browser, making API calls to the FastAPI backend for RDKit computations. This architecture keeps the frontend lightweight while leveraging the existing Python/RDKit infrastructure for molecular analysis.

```
# Build and serve cd standalone trunk build --release python serve.py # Serves on
http://localhost:8001
```

7.1.1 Molecular Property Scoring

The standalone app displays 10 molecular properties with color-coded scoring:

Property	Description	Good Range	Color Coding
QED	Drug-likeness (0-1)	> 0.5	Green > 0.5, Yellow > 0.3, Red ≤ 0.3
SA Score	Synthetic accessibility (1-10)	< 3.0	Green < 3, Yellow < 5, Red ≥ 5
Mol. Weight	Molecular weight (Da)	—	Neutral
LogP	Lipophilicity	—	Neutral
TPSA	Topological polar surface area (Å²)	—	Neutral
H-Bond Acc.	Hydrogen bond acceptors	—	Neutral
H-Bond Don.	Hydrogen bond donors	—	Neutral
Rot. Bonds	Rotatable bonds	—	Neutral
Rings	Ring count	—	Neutral
Heavy Atoms	Non-hydrogen atoms	—	Neutral

Table 11: Molecular properties computed by the AI-LNP Agent.

7.2 Desktop Application (wry + tao)

The desktop application is a native executable built with Rust using wry 0.47 (a cross-platform WebView library) and tao (a window management library). It embeds the production build of the Angular frontend at compile time using Rust's include_str! and include_bytes! macros, eliminating the need for external file dependencies. The entire application—HTML, JavaScript, CSS, and assets—is compiled into a single

binary (~8 MB) that runs on Linux, macOS, and Windows without requiring a web browser or Node.js runtime.

The desktop app uses a custom protocol handler (app://localhost/) to serve the embedded Angular files. When the user navigates to Angular routes (e.g., /reactions, /workflow), the protocol handler implements SPA fallback by serving index.html for all unmatched paths, ensuring client-side routing works correctly. The application opens in a 1280x820 window with the title '■■ Reactome LNP Agent' and connects to the FastAPI backend on localhost:8000.

```
# Build desktop app cd standalone-desktop ./build.sh # Compiles to
target/release/lnp-desktop # Run (requires backend on port 8000)
./target/release/lnp-desktop
```

7.2.1 Embedded Assets

The desktop build process copies the Angular production build into standalone-desktop/frontend/ and embeds four files:

File	Type	Size	Purpose
index.html	HTML	~2 KB	SPA shell with script/style tags
main-*.js	JavaScript	~800 KB	Angular application bundle
styles-*.css	CSS	~50 KB	Tailwind CSS styles
favicon.ico	Icon	~15 KB	Window icon

Table 12: Embedded assets in the desktop application.

7.3 Deployment Comparison

Feature	Angular Web	Dioxus Standalone	Desktop (wry)
Runtime	Browser + Node.js	Browser only	Native (no browser)
Backend Dependency	Required (port 8000)	Required (port 8000)	Required (port 8000)
Build Size	~1 MB (dist/)	~200 KB (WASM)	~8 MB (binary)
Functionality	Full (chat + analysis)	Molecular analysis only	Full (chat + analysis)
Installation	npm install + ng serve	trunk build + Python	Single binary
Cross-platform	Yes (browser)	Yes (browser)	Yes (Linux/macOS/Win)

Table 13: Comparison of the three frontend deployment options.

8. Project Structure and Deployment

The project follows a monorepo structure with backend and frontend code colocated under src/. Jupyter notebooks for experimentation and testing are kept in notes/, while generated documentation lives in docs/. The FAISS index is persisted in data/ alongside the source documents, enabling fast startup without re-indexing. Two additional directories (standalone/ and standalone-desktop/) contain the Dioxus WASM app and wry desktop app respectively.

```
chem-agent/ ■■■ .env # AWS Bedrock configuration ■■■ pyproject.toml # Python dependencies (uv) ■■■ uv.lock # Locked dependency versions ■■■ run.sh # Start both servers ■■■ data/ ■ ■■■ papers/ # Research PDFs (3 + 30 files) ■ ■ ■■■ related papers/ # 30 additional papers ■ ■■■ lnp_data/ # Rules, reactions, CSVs ■ ■■■ faiss_lnp_index/ # Persisted FAISS index ■■■ src/ ■ ■■■ backend/ ■ ■ ■■■ config.py # .env loader ■ ■ ■■■ rag.py # FAISS + document ingestion ■ ■ ■■■ agent.py # LangGraph 5-node workflow ■ ■ ■■■ main.py # FastAPI (6 endpoints) ■ ■■■ frontend/reactome-ui/ # Angular 21 project ■■■ standalone/ # Dioxus WASM app (AI-LNP Agent) ■ ■■■ src/main.rs # Dioxus component ■ ■■■ Cargo.toml # Rust dependencies ■ ■■■ Dioxus.toml # Trunk config ■ ■■■ serve.py # Python HTTP server (port 8001) ■■■ standalone-desktop/ # wry desktop app ■ ■■■ src/main.rs # wry + tao window ■ ■■■ Cargo.toml # Rust dependencies ■ ■■■ frontend/ # Embedded Angular build ■ ■■■ build.sh # Build script ■■■ notes/ # Jupyter notebooks ■■■ docs/ # This document + diagrams
```

8.1 Running the Application

The application requires two processes: the FastAPI backend (port 8000) and the Angular dev server (port 4200). For remote development via VS Code, ports are forwarded through the SSH tunnel.

```
# Backend unset CUDA_VISIBLE_DEVICES .venv/bin/uvicorn src.backend.main:app --host 0.0.0.0 --port 8000 --reload # Frontend cd src/frontend/reactome-ui && ng serve --host 0.0.0.0 --port 4200 # Or both at once ./run.sh # Standalone Dioxus app (port 8001) cd standalone && trunk build --release && python serve.py # Desktop app (requires backend on port 8000) cd standalone-desktop && ./build.sh && ./target/release/lnp-desktop
```

8.2 Remote Access via VS Code

- Forward ports **8000** and **4200** in the VS Code Ports panel
- Access frontend at **http://localhost:4200** in local browser
- Backend Swagger UI available at **http://localhost:8000/docs**
- Jupyter notebooks can be run directly in VS Code with the remote kernel