Cognifix: Multimodal Multi-Agent

---

1. Introduction

Cognifix is an advanced **multimodal, multi-agent orchestrator** developed using **Kotlin** and **Jetpack Compose**. It unifies text, audio, image, video, and file-based inputs into a **single conversational interface**, where each input type is intelligently routed to specialized **domain agents** such as Finance, Research, Weather, or Legal.

Each agent functions autonomously yet cooperatively — retrieving, reasoning, and aggregating information before passing it to a central **LLM-based orchestrator** (powered by Gemini models) that synthesizes the final response.

Cognifix's architecture blends **agentic reasoning**, **function-calling orchestration**, and **API integration** into a modular Kotlin framework — demonstrating a scalable and extensible system that can handle heterogeneous data streams.

---

2. Development Environment

Cognifix is built using:

- **Language:** Kotlin
- **Framework:** Jetpack Compose (for declarative UI)
- **Build System:** Gradle (KTS configuration)
- **LLM Integration:** Gemini API (for multimodal reasoning and function calling)
- **Architecture Pattern:** Modular, Agent-Oriented Design

---

3. Project Structure Overview

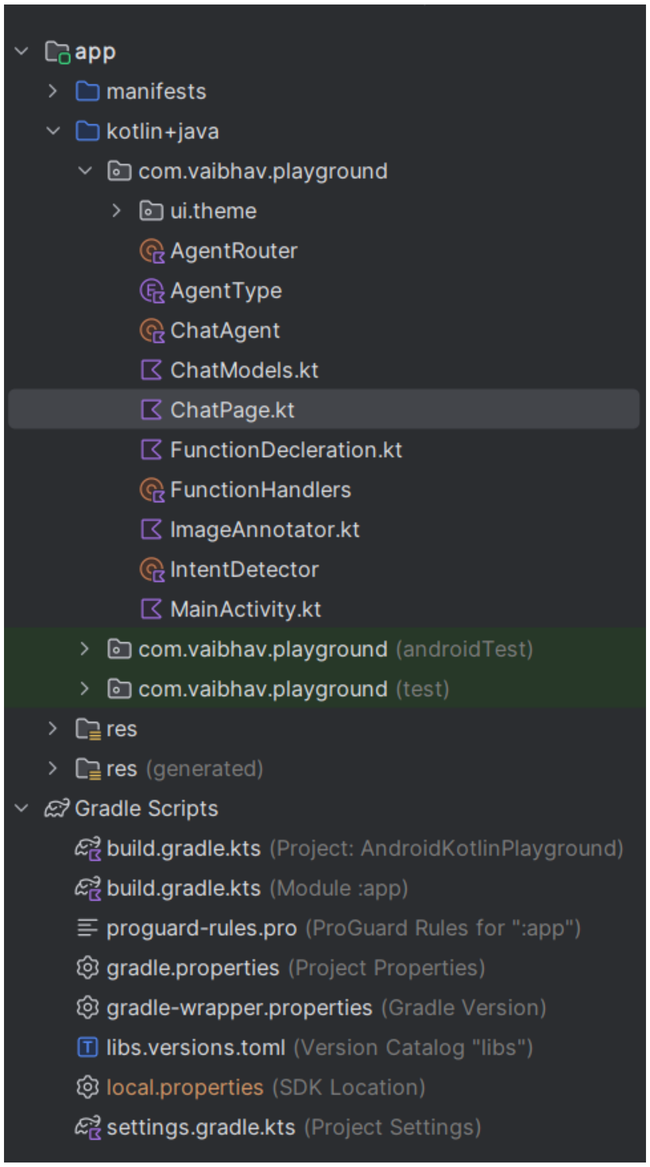
Below is the full Android Studio project hierarchy:

The structure is designed around **clean modular separation**, ensuring that UI, orchestration, and reasoning layers are isolated but interoperable.
This modularity allows rapid iteration, debugging, and scalability as new agents or modalities are added.

---

4. Directory Breakdown

4.1. Kotlin Source Directory



```
app
  manifests
  kotlin+java
    com.vaibhav.playground
      ui.theme
      AgentRouter
      AgentType
      ChatAgent
      ChatModels.kt
      ChatPage.kt
      FunctionDecleration.kt
      FunctionHandlers
      ImageAnnotator.kt
      IntentDetector
      MainActivity.kt
    com.vaibhav.playground (androidTest)
    com.vaibhav.playground (test)
  res
  res (generated)
Gradle Scripts
  build.gradle.kts (Project: AndroidKotlinPlayground)
  build.gradle.kts (Module :app)
  proguard-rules.pro (ProGuard Rules for ":app")
  gradle.properties (Project Properties)
  gradle-wrapper.properties (Gradle Version)
  libs.versions.toml (Version Catalog "libs")
  local.properties (SDK Location)
  settings.gradle.kts (Project Settings)
```

Main Package: com.vaibhav.playground

The Kotlin source directory forms the logical backbone of *Cognifix*. Each file has been designed to encapsulate a distinct concern — whether it's intent detection, routing, orchestration, or agent reasoning. This modularity makes the system easy to extend, debug, and maintain.

File Overview

File	**Description**
MainActivity.kt	Serves as the application's **entry point**. It initializes the Jetpack Compose environment, sets up runtime permissions, and connects the UI layer (ChatPage.kt) with backend orchestrator components.
ChatPage.kt	The **primary composable screen** responsible for rendering the multimodal chat interface. Handles user inputs such as text, audio, images, or files, and forwards them to ChatAgent.kt for processing. It also maintains live UI state using Compose's reactive data model.
ChatModels.kt	Defines **structured data classes** for both user messages and agent responses. It ensures consistent serialization and deserialization of multimodal inputs—ChatItem represents user input types (text, image, file, etc.), while AgentResponse captures model outputs (text, media, or links). This structure guarantees uniform data flow across UI, orchestrator, and LLM layers.
AgentRouter.kt	The **intent dispatcher** that classifies user input and routes it to the correct domain agent. It employs lightweight semantic checks (keyword and phrase mapping) and uses AgentType.kt to ensure type-safe agent invocation.
AgentType.kt	An **enumeration of domain agents** such as FINANCE, RESEARCHER, TRAVEL, and DESIGNER. Acts as a contract for consistent routing, allowing new domains to be added by simply extending this file.
ChatAgent.kt	The **core reasoning module** of Cognifix. Manages the lifecycle of each chat session, persistent memory, and real-time communication with the Gemini 2.5 Flash model via Firebase AI. It supports multimodal streaming responses, executes function calls (tools), and integrates with FunctionHandlers.kt to perform API calls. This file effectively bridges **user intent → agent reasoning → LLM orchestration**.
IntentDetector.kt	Implements **contextual NLP-based intent detection**. It scans text and media for trigger phrases (e.g., "highlight," "fix," "compare") to determine whether a visual editing pipeline (ImageAn) or a standard text agent should handle the query. It also classifies sub-intents like *fix*, *explain*, *creative*, or *compare*.
FunctionDeclaration.kt	Declares all **function signatures exposed to Gemini's function-calling interface**. These declarations describe callable tools (e.g., fetchWeather, fetchStockData, getCoordinates) and act as the bridge between model reasoning and real-world API execution.

File	Description
FunctionHandlers.kt	Contains the **actual implementations** of all registered functions. Handles HTTP requests to external APIs—such as OpenWeather, Financial Modeling Prep, or Serper Search—processes their responses, and returns clean JSON back to the orchestrator. It ensures that data retrieval and LLM reasoning remain decoupled.
ImageAnnotator.kt	Provides **multimodal image-processing capabilities**. Performs OCR extraction, annotation, and contextual markup of images or documents before passing them to the model. Enables Cognifix to handle visual reasoning, document summarization, and diagram interpretation tasks.

4.2. ui.theme Directory

Holds theming, typography, and color definitions for the Jetpack Compose UI, ensuring visual consistency and adherence to Material Design principles.

4.3. res Directory

Contains static assets such as drawable icons, string resources, and layout files required by non-Compose elements.

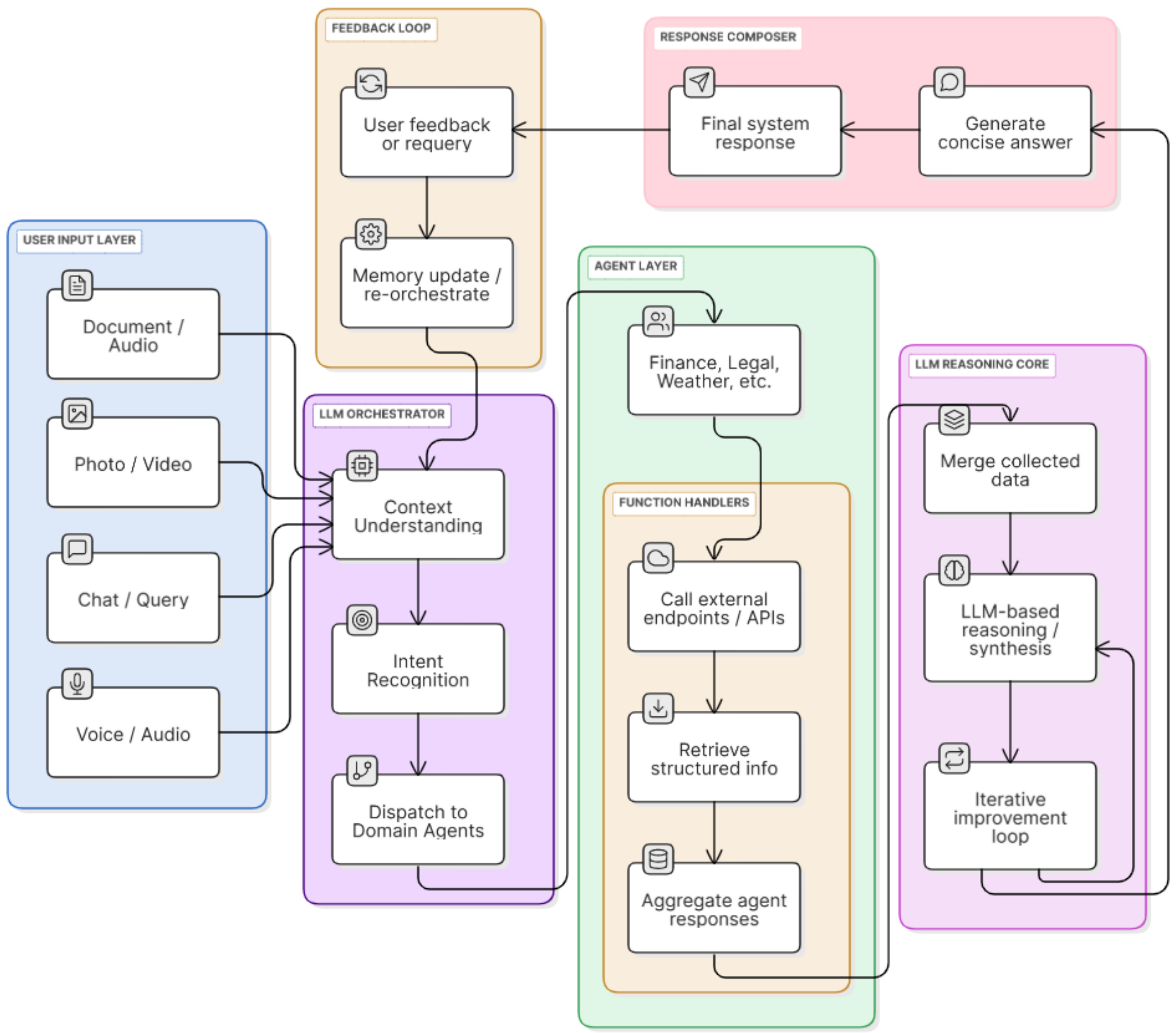
4.4. Gradle Scripts

File	Function
build.gradle.kts (Project)	Configures plugin dependencies, repositories, and build versions.
build.gradle.kts (Module: app)	Manages app-specific dependencies including Compose, Retrofit, and Gemini SDK integrations.
settings.gradle.kts	Registers project modules and dependency repositories.
gradle.properties	Defines project-wide variables, API keys, and build flags.
libs.versions.toml	Centralized dependency version management using Gradle Version Catalog.
proguard-rules.pro	Contains rules for code optimization and obfuscation during release builds.
local.properties	Configures the local Android SDK path and sensitive environment variables.

---

5. Agent Interaction and Architecture

The multi-agent orchestration architecture is summarized below:



5.1. User Input Layer

Users can interact with Cognifix through multiple input modalities:

- **Text or Chat** — via the Compose UI (ChatPage.kt)
- **Audio or Voice Commands** — recognized using the Android speech API
- **Image or Video** — processed through ImageAnnotator.kt
- **Document Uploads** — converted to text and parsed for intent detection

Each input is transformed into a structured ChatItem object before being passed to the orchestrator.

5.2. LLM Orchestrator

The orchestrator (composed of AgentRouter.kt and IntentDetector.kt) performs:

1. **Context Understanding** – Extracts semantic intent and relevant parameters.
2. **Intent Recognition** – Determines which domain(s) are relevant.
3. **Dispatching** – Routes the processed input to corresponding domain agents.

This design abstracts the LLM's reasoning layer from raw input, ensuring clean separation between understanding and execution.

5.3. Agent Layer

The **Agent Layer** (defined in ChatAgent.kt and AgentType.kt) handles specialized reasoning:

- Each agent encapsulates domain-specific logic (Finance, Legal, Research, Weather, etc.).
- Agents may work in isolation or in collaboration, depending on user intent.
- Communication between agents follows a message-passing protocol, enabling collective reasoning.

5.4. Function Handlers Layer

FunctionHandlers.kt bridges the gap between Cognifix's internal agents and external data sources:

- Executes REST API calls (e.g., stock price, weather data, or translation APIs).
- Parses JSON responses into structured Kotlin data objects.
- Returns consistent, validated outputs to be used by the reasoning layer.

This modular design allows easy integration of new tools or APIs by simply adding a new handler function and registering it in FunctionDeclaration.kt.

5.5. LLM Reasoning Core

The **LLM Core** orchestrates the final reasoning cycle:

1. Aggregates structured results from domain agents.
2. Performs high-level reasoning or synthesis using the Gemini model.
3. Generates concise, human-readable responses.
4. Implements an iterative loop for clarification or refinement.

This ensures that the final output is not just a data dump but a synthesized explanation derived from multiple knowledge sources.

5.6. Response Composer

The **Response Composer** (implemented in ChatAgent.kt and UI-handled by ChatPage.kt) generates coherent and contextually relevant responses:

- Converts structured reasoning results into conversational output.
- Handles formatting, tone, and contextual memory.
- Supports continuity across turns in a dialogue.

5.7. Feedback Loop

After each response, Cognifix enters a **Feedback Loop**:

- Collects user feedback or re-queries.
- Updates conversational memory or re-triggers orchestration.
- Allows adaptive refinement in case of incomplete or incorrect responses.

This feedback-driven design gives Cognifix a self-correcting, learning-oriented behavior without explicit model retraining.

---

6. Kotlin File Interaction Flow

The Cognifix architecture follows a **layered, event-driven interaction pipeline**.
Each Kotlin component plays a well-defined role in the transition from **user input** to **final multimodal output**, ensuring separation of concerns, clear data boundaries, and easy extensibility.

---

1. Initialization and Environment Setup

- **MainActivity.kt** serves as the application's entry point.
  It initializes the Jetpack Compose runtime, sets up permission handlers (for camera, mic, and storage access), and launches the root composable screen (ChatPage.kt).
- The initialization step also connects persistent memory instances, Gemini model configurations, and the Firebase AI runtime.
  Once setup is complete, the application is ready to capture multimodal inputs.

Flow:
MainActivity.kt → ChatPage.kt (UI composition and environment setup)

---

2. Multimodal Input Acquisition

- **ChatPage.kt** acts as the **frontline interface** for user interaction.
  It captures user messages (text, images, videos, audio, and files) through Compose components and converts them into unified objects defined by ChatModels.kt.
- Each input—whether a typed query, spoken command, or uploaded file—is wrapped as a ChatItem (e.g., ChatItem.Text, ChatItem.Image, ChatItem.Audio).

Flow:
ChatPage.kt → ChatModels.kt (generate ChatItem objects)

---

3. Intent Detection and Contextual Understanding

- The aggregated ChatItem list is analyzed by **IntentDetector.kt**, which uses keyword heuristics and lightweight NLP cues to determine whether the query involves visual editing, repair, explanation, or general reasoning.
- If the message includes both text and image, IntentDetector.kt can trigger specialized handlers such as NanoBananaHandler or ImageAnnotator.kt for multimodal enhancement.

Flow:
ChatPage.kt → IntentDetector.kt → (optional) ImageAnnotator.kt

---

4. Agent Routing and Domain Classification

- Once the intent is identified, **AgentRouter.kt** maps it to a specific **domain agent** (e.g., FINANCE, RESEARCHER, TRAVEL).
  This classification relies on AgentType.kt, which provides an enumeration of all available agent categories.
- AgentRouter.kt also attaches a **system prompt** template for the identified agent, allowing each domain to maintain a unique reasoning context.

Flow:
IntentDetector.kt → AgentRouter.kt → AgentType.kt

---

5. Agent Reasoning and Orchestration

- **ChatAgent.kt** takes control of the conversation flow.
  It initializes or resumes a chat session using the Gemini 2.5 Flash model and embeds the agent's system prompt for contextual grounding.

- The user's ChatItem payload is serialized into Firebase AI's Content format and streamed to the model. ChatAgent.kt supports **function calling**, enabling the model to trigger defined tools dynamically.
- The agent maintains persistent history, allowing multi-turn reasoning and contextual continuity between messages.

Flow:
AgentRouter.kt → ChatAgent.kt (model orchestration and reasoning)

---

6. Function Declaration and API Invocation

- When Gemini issues a tool-call request (e.g., fetchWeather, fetchStockData), **FunctionDeclaration.kt** provides the function signature and schema definition used to validate the call.
- **FunctionHandlers.kt** then executes the actual API logic, fetching structured data from external sources (OpenWeather, Financial Modeling Prep, Serper Search, etc.).
- Results are serialized as JSON and returned back to the model, closing the function-calling loop.

Flow:
ChatAgent.kt → FunctionDeclaration.kt → FunctionHandlers.kt → External APIs → ChatAgent.kt

---

7. Reasoning, Aggregation, and Final Response

- Once data is retrieved, **ChatAgent.kt** requests Gemini to synthesize and summarize all information into a concise, human-readable response.
- The orchestrator ensures the message aligns with the agent's domain tone — informative for Finance, conversational for Therapist, or creative for Designer.
- This reasoning cycle represents Cognifix's **agentic intelligence core**, combining structured retrieval with natural-language synthesis.

Flow:
FunctionHandlers.kt → ChatAgent.kt (synthesis and response composition)

---

8. Output Rendering and UI Update

- The synthesized agent response is streamed back to the **ChatPage.kt** interface.
  Depending on output type, the UI updates in real time with formatted markdown, text, images, or file links.
- Context history is maintained locally to support **multi-turn conversation** continuity.
  Users can view intermediate steps (e.g., API calls, annotations) as live progress indicators within the chat feed.

Flow:
ChatAgent.kt → ChatModels.kt → ChatPage.kt (UI rendering)

---

9. Feedback and Iterative Refinement

- The system optionally captures **user feedback** or clarification queries.
  This feedback is reintegrated into the context window through memory update functions, allowing adaptive reasoning in subsequent turns.
- This forms a **closed feedback loop**, enabling dynamic improvement of responses without re-initializing the conversation.

Flow:
ChatPage.kt → ChatAgent.kt (context update and memory loop)

Integrated Data–Control Pipeline

MainActivity.kt

↓

ChatPage.kt

↓

ChatModels.kt

↓

IntentDetector.kt

↓

AgentRouter.kt → AgentType.kt

↓

ChatAgent.kt

↔ FunctionDeclaration.kt

↔ FunctionHandlers.kt

↔ External APIs

↓

ChatModels.kt

↓

ChatPage.kt (render response)

This **cyclical, modular pipeline** enables Cognifix to process multimodal data, perform agent-specific reasoning, invoke APIs dynamically, and deliver context-aware responses — all within a unified Kotlin-based ecosystem optimized for integration within the Samsung AI framework.

---

7. Design Principles

The design of **Cognifix** follows clear **software engineering principles** to ensure modularity, scalability, and ease of integration. Each component — from intent detection to response generation — is encapsulated in a distinct Kotlin file, making the system highly maintainable and adaptable to change.

Below is a breakdown of the key architectural philosophies and how they are implemented:

7.1. Separation of Concerns

Cognifix's codebase is organized into distinct functional units, each responsible for one well-defined task:

- **Intent Detection** (IntentDetector.kt): Handles linguistic parsing, keyword extraction, and semantic classification.
- **Routing and Orchestration** (AgentRouter.kt): Maps recognized intents to appropriate domain agents and manages their execution.
- **Agent Logic** (ChatAgent.kt, AgentType.kt): Encapsulates domain-specific reasoning for finance, research, travel, etc.
- **API Integration** (FunctionHandlers.kt, FunctionDeclaration.kt): Connects agents to external services through modular and easily replaceable function definitions.
- **User Interaction** (ChatPage.kt, ChatModels.kt): Manages input/output rendering, maintaining the chat flow and UI responsiveness.

By isolating logic into these distinct Kotlin files, each component can be developed, tested, and debugged independently.
This **clear boundary between UI, logic, and orchestration** minimizes side effects and allows parallel development across different modules.

7.2. Modularity and Maintainability

Each Kotlin class or file in Cognifix serves as a **self-contained module**.
This modular design enables:

- **Independent Debugging:** A malfunction in API handling does not affect the chat UI or intent recognition. For instance, if an external API fails, the orchestrator can still function using fallback reasoning.
- **Ease of Testing:** Individual modules can be unit-tested without invoking the full app pipeline.

- **Version Control Simplicity:** Developers can modify or rollback specific agents or functions without introducing regressions in unrelated modules.

The use of **Jetpack Compose** further enhances maintainability — UI elements are reactive, declarative, and decoupled from business logic, ensuring a clear separation between presentation and computation layers.

7.3. Scalability

Cognifix is designed to scale both **horizontally** (adding new agents) and **vertically** (expanding capabilities of existing agents):

- **Adding New Agents:**
  To introduce a new domain (e.g., Healthcare or Education), developers simply add a new entry in AgentType.kt, create a corresponding handler class in ChatAgent.kt, and optionally extend FunctionHandlers.kt with new endpoints. No core architectural changes are required.
- **Adding New Modalities:**
  New input types (e.g., EEG signals, sensor data, AR inputs) can be integrated by extending the ChatModels.kt and updating the parsing layer — without disrupting the existing chat orchestration.
- **Distributed Processing:**
  Cognifix's multi-agent pipeline can run agents concurrently or sequentially depending on context, allowing future parallelization and server-based scaling.

This scalable architecture ensures that as new technologies, APIs, or domains emerge, they can be integrated with minimal reconfiguration.

7.4. Extensibility and Integration Flexibility

Cognifix was developed in **Kotlin**, which offers **seamless interoperability** with Java and strong compatibility within the **Samsung ecosystem**.
This makes the app easily portable and integrable across Samsung's hardware and software stack — from Galaxy smartphones and tablets to **SmartThings**, **Bixby**, and **WearOS** platforms.

Key aspects of its extensibility include:

- **Replaceable APIs:**
  Each external dependency (e.g., weather, finance, summarization) is abstracted through a handler. Replacing or upgrading an API requires modifying only its corresponding function — not the entire system.
- **Replaceable LLMs:**
  The orchestration logic is model-agnostic. Gemini APIs are currently used, but the same function-calling structure supports other models like **OpenAI**, **Anthropic Claude**, or **Samsung Gauss** without architectural modification.

- **Replaceable UI Layer:**
  Since the backend orchestration is decoupled from the Compose UI, it can be ported to alternative frontends — such as Flutter, SwiftUI, or web interfaces — with minimal effort.

This modular integration approach allows Cognifix to evolve rapidly as new APIs or LLM technologies emerge.

7.5. Debugging and Upgradability

Cognifix's file-based modular structure allows **granular debugging and quick upgrades**:

- Each file logs structured data during execution, making it easier to isolate which layer (intent detection, routing, or reasoning) failed.
- The use of data classes in ChatModels.kt ensures strict type safety, minimizing runtime errors.
- Any faulty agent or function can be disabled dynamically during runtime without affecting the rest of the system.
- Versioning at the file level enables clean rollback and independent code updates, ideal for CI/CD workflows or collaborative development.

This approach mirrors production-grade system design — modular, testable, and maintainable — allowing the platform to grow sustainably as features are added.

7.6. Adaptability and Future-Proofing

The **feedback loop** built into Cognifix ensures continuous refinement:

- Users can requery, correct, or expand on previous responses.
- The system captures feedback to re-orchestrate the agent reasoning sequence dynamically.
- This adaptability allows the orchestrator to improve its reasoning patterns over time, even without explicit model fine-tuning.

Moreover, the model-agnostic orchestration layer ensures **long-term sustainability**. As LLMs evolve, Cognifix's architecture can adapt by simply swapping out model connectors, preserving its high-level reasoning and orchestration pipeline.

7.7. Summary

In summary, Cognifix's architecture embodies:

- **File-level modularity** — clear separation of UI, logic, orchestration, and APIs.
- **Kotlin-native flexibility** — allowing smooth integration within Samsung's developer ecosystem.
- **Plug-and-play expandability** — agents, APIs, or models can be replaced independently.

- **Scalable orchestration** — easily extendable for more agents or input modalities.
- **Maintainable structure** — debug-friendly and version-control-safe.

Cognifix is thus not just a multimodal chat application, but a **scalable framework** for orchestrating AI reasoning across dynamic, evolving domains.

---

8. Conclusion

Cognifix represents a **modular and intelligent orchestration framework** capable of handling multimodal data and multi-agent reasoning within a native Kotlin environment.
By leveraging Gemini’s tool orchestration capabilities, it bridges human-like reasoning with structured domain knowledge, providing a scalable foundation for next-generation multimodal assistants.