

Umicom Studio IDE Revamp: Engineering Plan and Deep Dive

Introduction to Umicom Foundation and Ecosystem

The **Umicom Foundation** is a not-for-profit tech organization (based in Scotland) with a dual focus on humanitarian projects and open-source technology initiatives. One key project in its ecosystem is an educational content series (e.g. *Bits2Banking*) teaching computing from fundamentals to advanced topics. The foundation's projects span **content authoring**, **software development tools**, and **AI assistance**. The **Umicom Studio IDE** is envisioned as the central development environment tying these efforts together – a modern, GTK4-based integrated development environment for both coding and content creation tasks.

Umicom Studio IDE is not just a code editor; it integrates multiple components of the Umicom ecosystem:

- **Umicom AuthorEngine (UAEngine)** – a content authoring engine (with an AI assistant) to help produce books, documentation, and educational content.
- **Embedded LLM Interface** – local and cloud Large Language Model integration for AI assistance in code generation and content creation (using backends like llama.cpp for local inference, OpenAI API for cloud, and Ollama for local model serving).
- **Multi-Compiler Support** – a “compiler switcher” that allows building and running code in various languages (C, C++, assembly, Rust, Zig, etc.) using different compilers (TinyCC for quick lightweight C, GCC/Clang for full C/C++, Rust’s own compiler, Zig, and eventually a custom Umicom compiler “*Umicc*” in the future).
- **GTK4 & GtkSourceView UI** – a clean, modern GUI with code editing, project management, and possibly preview panes for content (leveraging GtkSourceView for syntax highlighting and code editing features).

The goal of this revamp is to take the current (broken) state of the Umicom Studio IDE repository and transform it into a **fully functional GTK4-based IDE**. This document provides a comprehensive plan: identifying existing issues, outlining the architecture, detailing build setup (especially for Windows), proposing repository restructuring, and defining milestones. It also includes step-by-step instructions and example LLM prompts to assist development of upcoming features. All source files will be consistently formatted and carry the MIT license header, in line with project guidelines.

Project Components and Integration Overview

To understand the engineering steps, it's important to break down the project into its sub-projects and how they will integrate within the IDE:

Umicom Studio IDE (GTK4 Application)

Umicom Studio IDE is the **GUI application** built with GTK4, acting as the front-end. It provides the user interface – windows, menus, editors, etc. – and orchestrates calls to backend components (compilers, UAEngine, LLMs). Key characteristics and tasks for the IDE component:

- **GTK4 and GtkSourceView UI:** The IDE's interface will use GTK4 widgets for the overall UI and **GtkSourceView** for the code editor component to provide syntax highlighting and code editing features.
- **Project/Document Management:** Manage different types of projects – code projects (C, C++, Rust, etc.) as well as content projects (for writing chapters, articles or books). Likely, the IDE will allow opening files of various types side-by-side.
- **Embedded Console/Output:** It should include a console output panel to show compiler messages, run output, etc. (This may come later in the roadmap but is a typical IDE feature).
- **Integration with UAEngine:** If the user is authoring content (e.g., a book chapter), the IDE might use UAEngine to build/preview the book or to call an LLM to generate text. For code projects, UAEngine's AI features might help in documentation or commit messages.
- **LLM Assistant Panel:** The UI might feature a sidebar or chat panel where the user can interact with an AI assistant (powered by the LLM backends) for code suggestions, explanations, or content generation.
- **Status and Current State:** Currently, the repository's GTK4 UI portion is incomplete. We see references to `src/app.c`, `src/window.c`, and a `studio_codestral_fim.c` in the CMake file, but these source files need review or might be missing. The build configuration suggests an executable called `ustudio` is to be built from these sources. Likely, `app.c` handles `GtkApplication` initialization, and `window.c` sets up the main window. `studio_codestral_fim.c` appears related to a "Fill-in-the-Middle" code generation demo (possibly using Mistral AI's *Codestral* model for code completion). We will verify and fix any missing pieces in these source files as part of the plan.

Umicom AuthorEngine (UAEngine) Module

The **UAEngine** is an AI-assisted authoring engine originally designed as a command-line tool (CLI). It helps with tasks such as initializing a writing workspace, ingesting content, building a draft (compiling Markdown to HTML/PDF via Pandoc), and even serving the output locally via a tiny web server. It also integrates AI features to assist in content creation. Key details:

- **Functionality:** UAEngine's CLI (`uaengine`) can create a boilerplate project (`init` command), combine chapters into a single Markdown draft (`build`), call Pandoc to export to HTML/PDF (`export`), open the result in a browser (`open`), serve via HTTP (`serve`), etc. The code indicates these commands and flow 1 2. This is tailored for book/documentation projects.
- **LLM Integration in UAEngine:** UAEngine includes an AI assistant component, abstracted behind `ueng_llm_*` functions. It allows connecting to different LLM backends. The environment variable `UENG_LLM_PROVIDER` selects the provider ("llama", "openai", or "ollama") 3 and corresponding settings (like model path or API keys in a config file or env variables) 4 5. In the current code, the OpenAI and Ollama backends are **stubs** (empty implementations) 6 7, while the **llama.cpp** backend has a partial real implementation (if compiled with `HAVE_LLAMA_H`) to load a local model and generate text 8 9 10. UAEngine even provides a `llm-selftest` command that opens a model and prompts it to "say hello" 11. This confirms that integration with an LLM is central, and one of the first uses was a self-test output.
- **Embedding UAEngine in the IDE:** To integrate UAEngine into the IDE, we will **refactor UAEngine into a reusable module**. Instead of the IDE invoking the UAEngine as a separate process, we can compile its code into a library or directly include the source in our build. Indeed, the repository already includes UAEngine's source files under `src/` (files like `common.c`, `fs.c`, `serve.c`, etc.), so likely the plan is to embed them. The CLI's `main()` can remain for

standalone use, but the IDE can call the underlying functions directly. We will ensure UAEngine's functionality (like building a book or calling the LLM) can be triggered from the GUI (e.g., via menu actions or buttons).

- **Status and Issues:** The UAEngine code provided is quite robust, with MIT license headers and extensive comments. However, we must identify if any **files are missing** from the repository. For example, `ueng/version.h` is included in `main.c` but it was not in the uploaded files – we should verify if it exists or needs generation (perhaps it defines `UENG_VERSION_STR`). We'll also ensure the UAEngine code compiles on all target platforms (there are already `_WIN32` guards throughout, which is a good sign for Windows compatibility). Any missing pieces (like the Pandoc invocation details or config files) will be noted.

Embedded LLM Interface (Local & Cloud)

A marquee feature of Umicom Studio is its **embedded AI assistant**, which can function both with local models and cloud APIs:

- **Local LLM via llama.cpp:** The IDE can run models directly on the user's machine using [llama.cpp](#) (or similar). The UAEngine code already expects a vendored `llama.cpp` (the code references an included `<llama.h>` and suggests vendoring it under `third_party/llama.cpp` with a CMake flag `UENG_WITH_LLAMA_EMBED`). When properly compiled with this, UAEngine's `ueng_llm_open/prompt/close` will use `llama.cpp`'s API to load a model (GGUF/GGML file) and generate completions **in-process**. This is ideal for offline use and performance. We need to ensure that during the build process, **llama.cpp** is included and built (or linked) so that `HAVE_LLAMA_H` is true and the real implementation is active. If not provided already, we will **add llama.cpp as a submodule or fetch it** in our build instructions. (We might fork `llama.cpp` to lock to a specific version or apply any minor patches).
- **Cloud LLM via OpenAI API:** For powerful models like GPT-4 or others, the IDE should allow using OpenAI's API. UAEngine's stub for OpenAI indicates how it is structured: we will implement it by calling the OpenAI REST API (via HTTPS). Likely we'll use **libcurl** for HTTP requests (the build already links to CURL). Our plan includes writing the code to `ueng_llm_open_openai` (to set up API key), `ueng_llm_prompt_openai` (to send the prompt and receive response), etc., and dispatching based on the `UENG_LLM_PROVIDER` selection. We should store the API key securely (probably in an environment variable or a config file loaded by UAEngine's config system).
- **Local Model Server via Ollama:** [Ollama](#) is an open-source tool that serves local models behind an API. UAEngine's Ollama stub suggests the intention to support it as well. To implement this, we'd use `libcurl` to call the Ollama HTTP endpoints (by default at `127.0.0.1:11434` for a local server) with the model name. For example, doing a POST to `http://127.0.0.1:11434/api/generate` with JSON including the prompt and model name. This provides a way to use sophisticated local models (like Mistral AI's **Codestral** code model, which Ollama supports) without linking them into our process. We will implement the Ollama backend similarly to OpenAI, differing mainly in endpoint and JSON payload structure.
- **Mistral Codestral Model & FIM:** The presence of `studio_codestral_fim.c` suggests an example or demonstration for code-specific LLM use. *Codestral* is Mistral AI's 22B code generation model that supports **Fill-in-the-Middle (FIM)** prompting. FIM means the model can take a prefix and suffix of code and generate the missing middle part. This is useful for advanced code completion scenarios. The "FIM demo" likely showcases how the IDE could send a prefix and suffix to the model (perhaps via Ollama, since Codestral can run there) and

•

then insert the model's output into the editor. Part of our plan is to **complete or correct the FIM demo integration**:

We will verify `studio_codestral_fim.c` content. It presumably uses libcurl to call an API (which explains the `CURL::libcurl` linkage needed). The CMake currently links `CURL::libcurl` to a target named `studio_fim_demo`, but that target wasn't explicitly created (likely a mistake). We'll fix the CMake so that either `ustudio` links to libcurl (if FIM code is part of main app) or define a separate `studio_fim_demo` target properly. If it's a separate demo program, we can decide to keep it separate or integrate its functionality into the main IDE.

- Implement the logic: Accept a code snippet with a gap, format an Ollama (Codestral) request with prefix and suffix (Ollama might require a special prompt format or API call with `fim`; the Mistral documentation suggests specialized endpoints for code completion ²²). The response would be the middle code to insert. This can greatly assist users writing code.
- **User Interface for LLM:** From a UX perspective, the IDE might present the LLM as a chat interface or as inline completions. Initially, we can implement a simple console-like panel or even use the **Output/Console window** for LLM interactions. For example, a user types a query or asks for code completion, and the assistant's response is displayed in a side panel. Eventually, more interactive UI can be added (like hovering suggestions or tab completions in the editor for FIM/code models).
- **Configuration:** The config system (via `UengConfig` struct) already supports user-specified defaults for LLM backends and model names ²³. We will expose these settings in the IDE's preferences (so the user can enter their OpenAI API key, choose a default local model path, etc.). UAEngine's `ueng_config_load_file` and `ueng_config_apply_env` can be leveraged to read a config file or environment variables ²⁴. We might provide a simple GUI dialog to edit these preferences and save to a yaml (flat key-value) file that UAEngine can parse.

To summarize, **getting the LLM interface fully functional** involves: vendoring and building llama.cpp, implementing the OpenAI and Ollama backends, and hooking up a UI for user interactions. This will transform current placeholder stubs into a powerful feature of the IDE.

Integrated Compiler Switcher and Multi-Language Support

To make Umicom Studio useful as a coding IDE, we will integrate support for compiling and running programs in multiple languages. The plan includes:

- **Supported Languages:** Initially focusing on **C** and **C++**, assembly, **Rust**, and **Zig**, since those were explicitly mentioned. Each language will have its own build/run pipeline:
- **C/C++:** We will allow switching between **TinyCC (TCC)** and **GCC/Clang** for compiling C/C++ sources. TCC is extremely fast and lightweight, useful for quick prototyping and possibly on-the-fly execution within the IDE (like a REPL or quick run). GCC (or Clang) will be used for full projects or when standard compliance and optimizations are needed. The user can choose the compiler in settings or per project. Eventually, **Umicc** (the Umicom C compiler, potentially developed by the foundation) will be integrated here as an option once it matures.
- **Assembly:** For assembly language source files, we can integrate with GCC as a driver (if using `.s` or `.asm` files with GCC) or use dedicated assemblers (e.g., **NASM** or GNU as). In the simplest case, if the assembly is in AT&T syntax, calling GCC on it might suffice. We'll include that in the build script. If writing raw assembly, we may bundle NASM for convenience or ask users to install it.

-
- **Rust:** The standard way to build Rust is via `rustc` or **Cargo**. We will not reimplement Rust compilation. Instead, the IDE will invoke `cargo build` or `rustc` as external processes. The compiler switcher in this case is trivial (Rust has one official compiler). But we will handle Rust-specific options (like edition, etc.) if needed. We should ensure Rust is installed (perhaps instruct user to install via `rustup`).
- **Zig:** Zig has its own compiler (`zig`) which can compile Zig code and also act as a C/C++ compiler. For Zig projects, we'll call `zig build` or `zig run` for single-file. Zig is standalone, so again user needs it installed (or we can consider bundling a version).
- **Executing Code:** After compilation, the IDE should allow running the program (if it's an executable and the target platform is the same). We'll implement a **Run** command that executes the compiled binary and captures its output in the IDE's console panel. For this, UAEngine's `exec_cmd` (or similar functions) can be repurposed or extended. In `common.c`, we see usage of `CreateProcess` on Windows or `system()` on POSIX^{25 26} for executing commands. We can use these facilities to run user programs, and redirect output to a GUI console (possibly by capturing `stdout` / `stderr` via pipes).
- **Design of Compiler Switcher:** We plan to create an abstraction in the IDE for "Build & Run" that picks the right toolchain based on file type or project settings. This could be a simple if/else mapping for now:
 - If the open file is `.c` or `.cpp`: use the selected C compiler (TCC or GCC).
 - If `.rs`: use `rustc/cargo`.
 - If `.zig`: use `zig`.
 - If assembly: use chosen assembler/GCC.
 - In future, if *Umicom's own compiler (Umicc)* is ready (possibly based on a subset of C or a new language), we will integrate it similarly. Umicc might even reuse parts of the above compilers or be built on LLVM. (Notably, the mention of **LLVM** in dependencies suggests that the future Umicc or other analysis features might rely on LLVM libraries for parsing or JIT. We will prepare for that by including LLVM build in our CI and possibly providing hooks for LSP or analysis using Clang libraries.)
- **Dependency Management:** For multi-file projects, building might involve linking multiple objects. Initially, we might assume single-file quick compile (especially with TCC). For more complex projects, we may later integrate a simple build system or utilize existing ones (e.g., Cargo for Rust handles multi-file, for C/C++ possibly use a CMake or makefile if project is bigger – perhaps beyond initial scope). Our **Milestones** will reflect that initial support is basic, with full project support in later phases.
- **User Interface:** A drop-down or settings panel in the IDE will list available compilers and allow selection. We'll autodetect if possible (e.g., check PATH for `tcc.exe` or `gcc`). A simple way is to require the user to set environment variables or point to compiler paths in a config. For example, an *IDE Settings* might have: *Preferred C Compiler: [TinyCC/GCC], Path to TinyCC: C:... \tcc.exe*, etc. We will supply sane defaults or detection (TinyCC can be bundled or built as part of our setup, see below).
- **Building Dependencies:** We need to ensure the compilers themselves are available:
- **TinyCC:** We can include TinyCC's source as a submodule (it's small and C89 code). Alternatively, instruct the user to build TinyCC. However, to make user's life easy, we might compile TinyCC as part of our build process (perhaps producing a `tcc.exe` we can package). The **TinyCC** codebase could be forked under Umicom for any necessary tweaks (for example, to support certain modern C features if needed, or just to ensure Windows compilation). As a Phase 1, we could

- provide instructions to manually install it, but integrating it into our CI to produce binaries would be great for a one-stop setup.
- **GCC/Clang:** On Windows, installing GCC usually means MinGW-w64 or MSYS2. Since we want an automated build environment, one approach is to rely on the user to have **MSYS2** and install `mingw-gcc`. Another approach is to use **LLVM/Clang** on Windows as the full compiler, since we are anyway including LLVM builds – Clang can compile C/C++ on Windows and link against the MSVC runtime, making it more self-contained than GCC. This might be why building LLVM was mentioned. We could ship an LLVM/Clang as part of the environment (though that's heavy). For now, we will document that either MinGW GCC or LLVM should be installed. The IDE's config can have a field for the path to `gcc` or `clang` as needed.
- **Rust/Zig:** We will not build Rust or Zig from source (these are big projects); instead we note that developers should install them via official channels (Rustup for Rust, download for Zig). Our instructions can include installing these tools as a prerequisite. In CI for Linux, we can easily `apt-get install rustc` or use actions/setup-rust. For Windows CI, we can use `choco install rust` or just run rustup.
- **Testing Multi-language:** We will create small example programs for each supported language to ensure the pipeline works. For example: a “Hello World” in C (with both TCC and GCC), a simple Rust program, a Zig snippet, and an assembly that prints a char. These can be used in a **“doctor”** or **“self-test”** command in the IDE similar to UAEngine's llm-selftest, to verify compilers are correctly configured.

With these measures, Umicom Studio will evolve from a content-centric tool to a lightweight but **polyglot IDE** suitable for education (where one might jump between writing explanatory text and example code in multiple languages).

Related Projects and Potential Integrations

The Umicom ecosystem doesn't exist in isolation – we can draw on or integrate with other open-source projects to accelerate development:

- **Nature Programming Language (nature-lang):** *Nature* is a new open-source language implemented in C, featuring an elegant syntax and its own compiler/runtime ^{27 28}. While not directly part of Umicom yet, it represents a path to having a custom language without heavy dependencies (it boasts not requiring LLVM and having a self-hosted compiler in pure C ^{29 30}). The foundation might consider adopting or forking *Nature* as a base for **Umicc** (the Umicom Compiler), or simply providing *Nature* language support in the IDE in the future. We suggest evaluating *Nature*'s compiler code (possibly including it as an optional toolchain in the long run) or at least ensuring the IDE's design can accommodate adding another language seamlessly (for example, adding an LSP for *Nature* or direct compile support since *Nature* can output machine code without external deps). This is a future possibility and can be a milestone after the initial IDE is stable.
- **C3 Language (c3lang):** C3 is an evolution of C with improved features. Its compiler (c3c) is open source ³¹. If Umicom's mission includes teaching system programming, supporting C3 could be interesting. We might not integrate it immediately, but the **repo layout and build system of C3** can inspire how to structure an out-of-tree language plugin. For example, C3 uses a separate project with multiple repos (compiler, standard library, etc.) ³². For Umicom Studio,

- adding support for C3 would involve embedding the c3 compiler or calling it. This again speaks to a plugin-like architecture for compilers which we can keep in mind.
- **Pygame Integration:** The Bits2Banking project heavily uses Python (over 90% per repository stats) so the IDE might eventually serve as a Python environment too. Pygame is mentioned likely because educational games or visual demos could be part of the curriculum. While the current focus is on C/Systems programming, we foresee a **phase where Python support is added**. This could mean embedding a Python interpreter to run scripts directly inside the IDE, or at least managing an external Python process. If we want to display graphics (like a Pygame window) initiated from the IDE, some work is needed so that the window either integrates or is handled properly (maybe just launch as separate window). **Integration suggestion:** We could create a mode where the IDE acts like an environment for Python – running a script and capturing output. Pygame-specific: if the IDE detects a Pygame program, it might auto-adjust (e.g., ensure the game window gets focus). These are refinements for later, but acknowledging them now ensures our architecture (particularly process execution and output capturing) is flexible enough for non-console programs as well.
- **OpenGL (Glad):** The mention of **Glad** suggests the potential need for OpenGL function loading in the C/C++ context. If Umicom Studio will have any feature that involves OpenGL rendering – for example, an integrated graphics/programming exercise where the IDE shows a GL output – we’d need an OpenGL loader. More straightforwardly, if users write OpenGL code, they typically use a loader like Glad or GLEW. We can assist by either bundling Glad or providing templates that include it. Perhaps the plan is to allow the IDE to run OpenGL examples and display a canvas (GTK4 has a `GtkGLArea` widget that could embed an OpenGL context). We propose **forking Glad** or using it as a subcomponent to generate a loader for the target platforms (Glad can be added as a C file to the project or built as a static lib). This integration would be helpful if we create a “graphics mode” or to support projects focusing on game dev or visualization.
- **LMStudio (Local AI Studio):** *LM Studio* is an application for running local LLMs via a nice UI and it provides SDKs in Python/TypeScript. While we are building our own IDE, we can take inspiration from LMStudio’s approach to model management. They likely handle downloading models, converting formats, etc. We **suggest integrating or forking parts of LMStudio’s backend**: for example, a **model manager module** that can download a model (like a GPT4All or Mistral model) if the user requests it, and then either use llama.cpp or call an API to run it. If LMStudio’s code is MIT or similar, we could even incorporate their CLI or library to avoid reinventing the wheel for things like model format handling. At minimum, referencing their documentation may help ensure we design our LLM config (model paths, etc.) in a user-friendly way. Additionally, projects like **LocalAI** (which provides a local OpenAI-compatible API) could simplify integration: rather than writing separate code for each backend, we could run a LocalAI server and point our OpenAI backend code at it. However, given we already will implement OpenAI and have Ollama for local, we likely stick to those for now.

In summary, Umicom Studio IDE will benefit from a modular structure that can accommodate multiple languages and tools. By considering these related projects, we ensure our design is forward-looking, allowing integration or optional support for new compilers, languages, and AI tools as the ecosystem grows.

Identifying Current Issues and Gaps

Before implementing new features, we must fix the **broken state of the repository**. Through analysis of the provided source files and build scripts, we pinpoint the following issues:

- **Build System Misconfigurations:** The `CMakeLists.txt` is incomplete or contains errors. It defines the `ustudio` executable and adds source files, but then links GTK4 only to `ustudio` and links libcurl to a target `studio_fim_demo` that was never created. This appears to be a mistake. We will correct this by either:
 - Merging the FIM demo code into `ustudio` (and thus linking libcurl to `ustudio`), or
 - Defining a separate `add_executable(studio_fim_demo ...)` if we want a standalone demo binary. Given that `studio_codestral_fim.c` might rely on GTK as well (if it's part of the GUI), it likely should be part of `ustudio`. So the simplest fix is: **include libcurl in `ustudio`'s link libraries** and remove/ignore the stray `studio_fim_demo` reference. This ensures any code using `curl` (OpenAI/Ollama calls, etc.) will link properly.
- **Missing Source Files:** We need to verify that all necessary source files are present:
 - `app.c` and `window.c` : These were referenced but not provided in the snippet. If they don't exist in the repo or are skeletal, we must implement them. Typically: `app.c` will create a `GtkApplication (g_application_new()` with an application ID) and in its activation handler, create the main window.

`window.c` will define the `GtkApplicationWindow`, set up menus, a `GtkSourceView` for the editor, etc. If these files are missing, the IDE can't even start, so creating or completing them is top priority. We'll likely create them anew with proper MIT headers and a basic UI layout (menubar, a paned window with an editor and maybe a sidebar for future use).

- `studio_codestral_fim.c`: The presence is known from CMake, but we should retrieve or recreate its content if missing. This file presumably contains code related to demonstrating fill-in-the-middle. If we cannot retrieve it, we will implement a new version that uses the Ollama backend to get a FIM completion (or at least outline how it *would* be done). We will add the MIT header and ensure it fits into the build.
- Header files: We saw `common.h`, `config.h`, etc., provided. But `version.h` was not provided. Possibly `version.h` defines some version string and might be auto-generated by CMake (common practice to generate a version header from Git tags or so). Since the code checks `#ifndef UENG_VERSION_STR` ³⁶, it likely expects `-DUENG_VERSION_STR="0.1.0"` to be passed, or the header defines a default. We will create a simple `version.h` if needed (with a placeholder version).
- Check for any other includes like `ueng/llm.h` which likely declares the LLM API and constants for backend IDs (we saw definitions in `llm_llama.c` for backend IDs ³⁷). If `llm.h` is missing, we must write it (with prototypes for `ueng_llm_open/prompt/close` and possibly some structures). The stub files include "`ueng/llm.h`" and rely on it for declarations ^{38 39}, so providing this header is necessary. We can reconstruct it from the implementations:

It should declare `struct ueng_llm_ctx` (opaque handle) and the three functions `ueng_llm_ctx *ueng_llm_open(const char *model_path_or_name, int context_tokens, char *err, size_t errsz), int ueng_llm_prompt(ueng_llm_ctx *ctx, const char *prompt, char *out, size_t outsz)`, and `void ueng_llm_close(ueng_llm_ctx *ctx)`. Additionally, any provider-specific open functions if needed, or an enum for provider type (like `UENG_LLM_BACKEND_LLAMA = 1`, etc.). We saw those defines in `llama.c` ⁴⁰.

We will add this header with the MIT header and ensure it's included where needed.

- **Linking and Libraries:** The project requires **GTK4** and **GtkSourceView** libraries, as well as **libcurl**. On Windows, these are not commonly installed, so building or providing them is part of the challenge:
- If any of these libraries are missing at build time, CMake will error. The CMake uses `pkg_check_modules(GTK4 REQUIRED gtk4)` which means it expects `gtk4.pc` to be found by pkg-config. We must ensure our environment has a .pc file for GTK4. Similarly for `GtkSourceView` (not explicitly in CMake, but we will add it for code editing component, e.g. `pkg_check_modules(GTKSOURCEVIEW4 REQUIRED gtksourceview-5)` if using version 5 for GTK4).
- On Windows, we might not have pkg-config by default; our build instructions will cover setting that up (possibly via `vcpkg` or `MSYS2` for dev files, or by building from source and installing).
- **Libcurl** is found by `find_package(CURL REQUIRED)`. CMake will find it if a libcurl is installed (on Windows maybe via `vcpkg` or a prebuilt binary). We'll handle this in our setup steps (possibly building curl from source or installing it).
- **CI and Cross-Platform Considerations:** As of now, the project isn't set up with CI. We need to create a CI pipeline that:
 - Compiles on **Windows** (since initial target is Windows, possibly using GitHub Actions with a Windows runner and PowerShell build steps).
 - Ensures Linux compilation (perhaps in a separate job using Ubuntu, where GTK4 can be installed easily via `apt`).
 - Prepares for **RISC-V**: This likely means cross-compiling for a RISC-V Linux target (since Windows on RISC-V isn't a mainstream thing, we assume RISC-V means Linux on RISC-V dev boards or

emulators). We might use a cross toolchain or QEMU. This is for the future, but we note it now: when we build dependencies, we should ensure they can be cross-compiled or have build scripts for RISC-V.

- Also, running tests: we might include running the UAEngine's `llm-selftest` in CI (with a small dummy model or a stub mode) to ensure the LLM integration stubs don't break the build.

By addressing these issues – fixing CMake, adding missing files, ensuring dependencies – we will bring the repository to a **buildable state**. With that foundation, we can proceed to implement new features and improvements as outlined.

Repository Restructuring for Clarity

To improve maintainability, we will restructure the repository layout in a logical way. A clear separation of modules will help contributors navigate the project and also allows us to add/remove components cleanly. Proposed changes:

- **Top-Level Directories:**
- `ide/` (or `ustudio/`): Contains the GTK4-based IDE application source code (main function, app/window, UI files, etc.).
- `uaengine/`: Contains the Umicom AuthorEngine module source code (which can be built as a static library or object files). This would include `common.c`, `fs.c`, `serve.c`, `llm_*.c`, `ueng_config.c`, etc., as well as an `include/` subdirectory for headers (`common.h`, `config.h`, `llm.h`, etc.).
- `third_party/`: Contains external dependencies we vendor. For instance, `third_party/llama.cpp` for LLM integration, possibly `third_party/tcc` for TinyCC source, `third_party/glad` for OpenGL loader, etc. Each may have its own license file if required. We will integrate their build into our CMake as needed (either via `add_subdirectory` or prebuilding).
- `data/`: Contains data files for the application. The existing `data/ustudio.gresource.xml` (used to compile embedded resources like icons, UI definitions, CSS) would live here. We will store our UI definition (if using GtkBuilder XML files for menus or dialogs) and icons here.
- `scripts/`: Any helper scripts (PowerShell or Bash for CI, etc.) can reside here. For example, a `scripts/bootstrap.ps1` to set up dependencies on Windows, or CI YAML files if not in `.github`.
- `docs/`: Documentation like README, design documents, etc. The markdown report (this document or its refined version) could be placed here for reference, along with any usage guides we write for end users.
- **CMake Build Structure:** Instead of one flat CMakeLists, we can have:
- A top-level CMakeLists that defines the project and includes subdirectories.
- `uaengine/CMakeLists.txt` to build a static library `libuaengine.a` (or on Windows a `.lib`). This will compile all `uaengine/*.c` files. The UAEngine `main.c` (CLI) could either be omitted in library build or put in a separate target.
- `ide/CMakeLists.txt` to build the `ustudio` executable. This will depend on `uaengine` (link against that static lib) and on GTK4, GtkSourceView, etc. Also link to `libcurl`. It will include `app.c`, `window.c`, etc.
- Optionally, an `uaengine_cli/CMakeLists.txt` if we want to build the standalone CLI `uaengine.exe` as well. That would just combine `main.c` with the `uaengine` library.
- This structure ensures the code is not duplicated, and it's clear which part is core logic vs UI.
- **Consistent Formatting and Headers:** We will add a `.clang-format` configuration (if not already) to enforce the "strict formatting". The Nature language repo had a `.clang-format` ⁴¹, we can adopt

similar style or define one for Umicom (most likely 4-space indent, certain brace style, etc.). We will run this on all source files. Additionally, every source and header file will start with the standardized MIT license header (as seen in provided files). We'll verify any new files we create (like `llm.h` or `window.c`) include:

```
/
*-----
* Umicom Foundation ...
* File: ...
* Purpose: ...
* Author: ...
* License: MIT
*----- */
```

to keep consistency.

- **Remove or Isolate Generated Files:** The build generates `ustudio-resources.c` from the XML file. We will keep generated files out of the source tree (the CMake already outputs it to `CMAKE_CURRENT_BINARY_DIR`). The `target_include_directories(ustudio PRIVATE $ {CMAKE_BINARY_DIR})` ensures that file is found. So that is fine. We just need to ensure the XML includes any new resource files (like icons or UI .ui files).
- **README and Documentation:** We will update the README to reflect the new structure and build instructions. A brief introduction to each sub-project (IDE, UAEngine, etc.) will be included so newcomers understand the repository layout quickly. We will also mention how this fits into the bigger Umicom ecosystem and where to find more info (like linking to umicom.foundation site or Bits2Banking for context).

This reorganization will make the repo cleaner and more approachable. New contributors can, for example, work on UAEngine in isolation or focus on the IDE UI without interfering with each other. It also sets the stage for possibly splitting the repo in the future if needed (UAEngine could become a standalone library project on its own, for instance). For now, it remains one repository but logically separated.

Step-by-Step Build Environment Setup (Windows focus)

One of the deliverables is a step-by-step guide to get the project building, especially on Windows using PowerShell. We outline the process here, which can also be adapted for CI:

Note: The instructions assume a Windows 10/11 64-bit system. Ensure you have **Visual Studio 2022** or the **Build Tools** installed for C/C++ (for MSVC compiler), as well as **CMake (>= 3.20)** on your PATH. We will use PowerShell commands. For Linux users, equivalent commands (using apt or yum for dependencies) will be provided in a later section or documentation.

1. Clone and Prepare the Repository

First, acquire the Umicom Studio IDE source code and enter the directory:

```
# Clone the repository
git clone https://github.com/umicom-foundation/umicom-studio-ide.git
cd umicom-studio-ide

# (Optional) Check out a specific branch or tag if needed
# git checkout -b development origin/development
```

If the repository uses submodules for dependencies (we plan to add some, like llama.cpp), ensure they are fetched:

```
# Initialize and update submodules (if any are configured)
git submodule update --init --recursive
```

At this point, the directory structure should have `ide`, `uaengine`, etc., as per our restructuring (if not yet applied in code, mentally note the locations of files from earlier layout).

2. Install/Fork and Build Dependencies from Source

Next, we need to set up all required libraries. On Windows, this can be the most complex step, since libraries like GTK and GtkSourceView are not trivial to build. We present two approaches: using vcpkg (for simplicity) or manual build (for full control). The instructions below lean toward **manual building** to satisfy the “fork and build” requirement:

2.1. Install Build Tools and Package Managers (if not already installed):

- **Meson and Ninja:** GTK4 uses Meson build system. Install via pip:

```
pip install meson ninja
```

- **pkg-config:** Needed for CMake to find libs. You can get it via MSYS2 or vcpkg. If using vcpkg later, it provides a toolchain that simulates pkg-config. Alternatively, use Chocolatey:

```
choco install pkgconfiglite
```

Ensure `pkg-config.exe` is in PATH.

- **Perl, Python:** Required by some dependency builds (Perl for Glib). Using MSYS2 for building GTK will include these. If doing manually, install Strawberry Perl and make sure python is available (already likely since we used pip).

2.2. Build GTK4 from source (Windows):

GTK4 on Windows has many dependencies (GLib, Cairo, Pango, etc.). The easiest route is to use MSYS2 to get a working build environment. However, since we want to fork and build, we might do this:

- **Fork/Download GTK 4 source:** Download the source bundle from the GNOME site or fork the repository. We need at least GTK 4.x and GtkSourceView 5.x.
- **Build Glib:** GTK’s core dependency is GLib. In an MSYS2 MinGW64 shell, one would normally do:

```
pacman -S mingw-w64-x86_64-gtk4 mingw-w64-x86_64-gtksourceview5
```

But to build manually:

- `meson setup builddir` and `meson compile` for glib.
- Then build GObject-Introspection, Cairo, Pango, etc.
- Finally build GTK4.

Given the complexity, a **pragmatic approach** for initial bring-up: use MSYS2 binaries. We can install the dev packages and then use their



```
# Install MSYS2 (if not already). Then in MSYS2 shell:
pacman -Syu # update
pacman -S mingw-w64-x86_64-gtk4 mingw-w64-x86_64-gtksourceview5 mingw-w64-x86_64-toolchain
pacman -S mingw-w64-x86_64-curl
```

.pc files. For instance:

This gives us pre-built GTK4, GtkSourceView, and Curl, plus GCC. After this, in a VS Developer Command Prompt or PowerShell, set:

```
$env:PKG_CONFIG_PATH = "C:\msys64\mingw64\lib\pkgconfig"
$env:PATH += ";C:\msys64\mingw64\bin"
```

Now `pkg-config --modversion gtk4` should return a version, confirming the environment is set.

However, if we interpret “fork and build” strictly: we might instead incorporate these dependencies into our build process (like building a custom static GTK). This is an advanced task that might be deferred. For immediate progress, **we will use MSYS2 or vcpkg** to get the libraries working, and later consider forking if customization is needed.

2.3. Build GtkSourceView: If using MSYS2, it's done above. If manual, similar Meson build as GTK. GtkSourceView provides the code editing widget.

2.4. Build CURL: libcurl is needed for HTTP requests. Since CMake `find_package` will pick it up, ensure it's installed: - Via MSYS2 we did (`mingw64-curl`). - Or vcpkg: `vcpkg install curl`. - Or build from source: download curl source, CMake configure for Win32/Win64 with OpenSSL or Schannel. But using a package is simpler.

2.5. TinyCC: For compiler integration, get TinyCC. We can fork the TinyCC repository (mob branch is recommended for latest fixes). Building TCC: - TCC on Windows can be built with MinGW. For now, we might skip building it ourselves and use a pre-built binary (to be placed in `third_party/tcc` or prompt user to download). If we want to integrate, we could compile it as part of our MSYS2 environment:

```
git clone https://repo.or.cz/tinycc.git
cd tinycc
./configure --config-msvc # they have some config for msvc or use ./
configure && make for gcc
make
```

Ensure `tcc.exe` is produced. We can copy that to our `bin` directory for the IDE or add to PATH.

Summary: The above steps are complex, so here is a streamlined path if using vcpkg (an alternative approach):

```
# Using vcpkg to install dependencies easily git
clone https://github.com/microsoft/vcpkg.git
.\vcpkg\bootstrap-vcpkg.bat

# Install GTK4, GtkSourceView, CURL via vcpkg
.\vcpkg\vcpkg.exe install gtk4:x64-windows-static gtk4:x64-windows # static or
dynamic
.\vcpkg\vcpkg.exe install gtksourceview5:x64-windows
.\vcpkg\vcpkg.exe install libcurl:x64-windows
```

This will download and build those libs. Then, when configuring CMake, use the vcpkg toolchain:

```
cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=path\to\vcpkg.cmake -
DVCPKG_TARGET_TRIPLET=x64-windows
cmake --build build --config Release
```

For our PowerShell instructions, we might prefer the vcpkg route because it's step-by-step and avoids diving into each library's build intricacies. We will document it with the understanding that in CI or advanced use, the foundation might maintain forks of these libs if needed.

3. Configure and Build Uicom Studio IDE

Assuming dependencies are in place (via MSYS2 or vcpkg as above):

```
# Create a build directory
mkdir build
cd build

# Configure the CMake project
cmake .. -G "Ninja" -DCMAKE_BUILD_TYPE=RelWithDebInfo `
-DGTK4_DIR="C:/msys64/mingw64/lib/pkgconfig" `
-DCMAKE_PREFIX_PATH="C:/msys64/mingw64"
```

```
# If using Visual Studio generator:
# cmake .. -G "Visual Studio 17 2022" -A x64 -DCMAKE_BUILD_TYPE=Debug

# Build the project
cmake --build . --config RelWithDebInfo
```

A few notes on the above:

- We used Ninja generator for simplicity. If using the VS generator, remove Ninja and specify the architecture.
- We pass `CMAKE_PREFIX_PATH` or other variables to help CMake find GTK if needed. When `pkgconfig` is in `PATH`, `pkg_check_modules(GTK4 REQUIRED gtk4)` should work, but sometimes on Windows we need to hint the prefix.
- If `vcpkg` toolchain is used, you'd do:

```
cmake .. -G "Ninja" -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=D:\vcpkg\scripts\buildsystems\vcpkg.cmake
cmake --build . --config Release
```

After a successful build, we expect `ustudio.exe` (on Windows) in something like `build/RelWithDebInfo/` or `build/ustudio.exe` for Ninja builds. Also, possibly `uaengine_cli.exe` if we enabled building the CLI.

4. Running and Testing the Application

Before running, ensure that GTK4 runtime is accessible (if using MSYS2, the DLLs in `mingw64/bin` need to be on `PATH`, which we added; if using `vcpkg` dynamic, the DLLs will be copied to the exe folder or you have to copy them).

```
# From the build directory:
./ustudio.exe
```

When launching the IDE for the first time: - You should see a window come up (likely just a blank editor area or a simple UI if minimal implementation). - If any DLL is missing, you'll get an error; ensure `libgtk-4-1.dll`, etc., are present. In MSYS2 case, they are in the path we set. In `vcpkg`, you might need to copy the runtime manually or use the `vcpkg` manifest mode to package.

Try opening the **Help -> About** or any placeholder menu to ensure GTK is responsive. We don't have actual features yet like LLM working or compile, those will come as we develop.

5. Verification & Self-Tests

We can run some basic checks: - UAEngine CLI (if built): run `uaengine.exe llm-selftest`. It should either perform the test or print that environment is not set⁴². For example, if no model is configured you might see `"UENG_LLM_MODEL not set."` or an error from llama. This at least shows the binary runs. The IDE could have a menu action "AI Self-Test" wired to call the same function. If not, we can run it via a temporary command. - The embedded web server: we can simulate the `serve` command of UAEngine inside the IDE

or CLI to ensure `serve.c` works (it should start a tiny HTTP server on localhost). This requires having some site content; skip unless we have a prepared workspace.

If all these steps succeed, we have a functional build environment!

(Optional) 6. Setting Up CI for Windows

Using the above steps, we can automate in CI (GitHub Actions):

We'd create a `.github/workflows/build-windows.yml` that runs on `windows-latest`, uses actions for `vcpkg` or `choco` to install dependencies, then executes similar commands: - Checkout code Install `vcpkg` dependencies or `MSYS2` environment - `cmake . .` and `cmake --build` - Perhaps archive the artifact or run tests.

Similar CI files can be written for Linux (install GTK via apt, etc.) and for a RISC-V job (maybe using a cross-compiler or QEMU). This ensures continuous integration and that future changes don't break the build on any platform.

(The full CI scripts are beyond the scope of this text, but this plan ensures we are ready to create them.)

Milestones and Roadmap to Usability

With the repository building, we outline a roadmap of milestones to go from this initial state to a fully usable and maintainable IDE:

- **Milestone 1: Build Fixes & Basic UI (Week 1-2)**

Goals: Clean build on Windows and Linux; basic GTK4 window opens.

Tasks: Fix CMake issues (linking, missing files), ensure all dependencies in place. Implement `app.c` and `window.c` with a simple menubar (File, Edit menus), an empty `GtkSourceView` in the window, and placeholders for future UI elements (like a sidebar for project or an output console at the bottom). Include About dialog with version (use `UENG_VERSION_STR`). *Outcome:* A developer can compile and launch Umicom Studio, see the main window and basic UI components (though they may not do much yet). The codebase has proper structure and headers. Establish CI for build-only validation.

- **Milestone 2: UAEngine Integration & Content Authoring (Week 3-4)** *Goals:*

Integrate UAEngine's functionality into the GUI for content projects.

Tasks: Expose actions in the UI that correspond to UAEngine commands. For example, "New Book Project" (calls UAEngine init to create workspace), "Build Draft" (calls pack and Pandoc export via UAEngine build/export), "Preview Site" (launches browser or embedded WebKit to show the generated site), "Serve Site" (spawn UAEngine's serve in background thread and open browser). This will require threading or async handling so the UI doesn't lock up during long operations (like Pandoc running). Also, allow configuring the LLM provider and model in preferences and use UAEngine's config to set environment accordingly.

Outcome: Non-coding users (like authors) can use the IDE to create and build content projects with AI assistance (for example, an "AI Suggest Paragraph" feature could feed the last lines of a chapter to `ueng_llm_prompt` and insert the response). On the code side, this milestone solidifies the integration of UAEngine library calls and ensures the GUI <-> UAEngine interface works.

• **Milestone 3: Code Editing and Compilation for C/C++ (Week 4-6)** *Goals:* Make

the IDE actually useful for coding in C/C++.

Tasks: Implement the compile & run commands. Provide a way to select a source file or project and compile it. We might start with single-file support: e.g., if a C file is open, enable a “Run” button that will compile it (via TCC by default) and then execute it, capturing output. Show output in a bottom console panel. If compilation fails, show errors (we can parse GCC/TCC error output minimally to jump to lines). Also implement the compiler selector UI and underlying logic to switch between TCC and GCC. This may involve setting environment or calling different commands. Possibly integrate with `slugify` or other UAEngine utils for naming outputs. *Outcome:* A user can write a simple C program in the IDE, click Run, and see the program output. They can toggle a setting to use GCC instead and see that it compiles (taking longer but allowing more complex code). This transforms the IDE into a basic but functional code IDE.

• **Milestone 4: Embedded LLM Coding Assistant (Week 6-8)** *Goals:* Provide AI-based code completion and chat within the IDE.

Tasks: Finalize the OpenAI and Ollama backend implementations. Add a UI panel (perhaps a sidebar titled “AI Assistant”) where the user can ask questions or get completions. For code completion (FIM), implement it such that when the user triggers it (maybe by a special comment or keyboard shortcut), the IDE takes the `current file buffer`, the `cursor position` as the split point, and asks the LLM to fill in code. Use `studio_codestral_fim` logic for this. The response should then be inserted into the editor. For the chat interface, handle it like a conversation: user prompt goes in, we call `ueng_llm_prompt` with that (perhaps with a system prompt instructing it to only provide helpful answer), then display the result. We have to manage streaming vs full response; initially, we can wait for full completion then display. Ensure that large outputs don’t freeze UI (maybe run in a worker thread).

Outcome: The IDE can act like modern AI-assisted IDEs (e.g., GitHub Copilot or GPT plugins), albeit with a simpler interaction. The user can get code suggestions or have the AI generate a function when asked. This milestone will greatly enhance the usability for learning (the AI can explain code or suggest improvements on the fly).

• **Milestone 5: Multi-Language Expansion (Week 8-10)** *Goals:* Add Rust and Zig support, and improve project handling.

Tasks: Allow creating/opening Rust projects (detect `Cargo.toml`, then enable “Build (Cargo)” and “Run” using `cargo run`). For Zig, if a `.zig` file is detected, allow running via `zig run`. Implement syntax highlighting for these languages (GtkSourceView typically includes definitions for many languages; ensure `.rs` and `.zig` are recognized or supply `.lang` files if needed). This milestone might also introduce *project templates* – e.g., File -> New Project -> [C Console App, Zig Console App, etc.] which scaffolds minimal code. We’ll also refine the build output parsing for these compilers if necessary.

Outcome: The IDE will no longer be limited to C – it can handle a variety of languages, making it more appealing to a broad range of users (especially in an educational context where different languages are explored).

• **Milestone 6: Polish, Documentation, and CI Enhancements (Week 10+)**

Goals: Solidify the project for public use and contributions.

Tasks: Write comprehensive **documentation** (in `docs/`) for developers and users. Cover how to build the project (largely derived from the steps above), how to add a new language or LLM backend (for contributors), and usage guides for end-users (how to set up API keys, etc.). Also establish **automated tests** for critical features if possible (for example, a small script to open the

app, run a known command, and verify output – perhaps using a CLI mode of the IDE for testing or just testing UAEngine functions directly). Expand CI to run tests and possibly produce binary packages. For instance, set up GitHub Actions to create a Windows installer or zip (with all needed DLLs and a copy of tcc, etc.) for each release. Ensure **MIT license compliance** by including a `LICENSE` file, and verifying that any third-party code we include (like llama.cpp or TCC) has compatible licenses (llama.cpp is MIT, TCC is LGPL – if we include TCC, we might need to handle LGPL compliance, perhaps by dynamically linking to TCC's lib or providing source). This milestone is about preparing for the future: making sure the foundation is solid for other developers to join in and for the project to scale.

- **Beyond:** After these, possible future steps include:
 - Integration of Umicom's own compiler *Umicc* if it emerges (perhaps Milestone 7).
 - Cross-platform GUI improvements (ensuring everything works on macOS as well, if that's a target).
 - Performance tuning, especially for LLM inference (maybe adding support for model quantization or using GPU, etc., which could be separate projects).
 - Community feedback and feature requests (maybe a plugin system, or debugging support for languages, etc.).

Each milestone builds upon the previous, bringing the project incrementally closer to the vision of a **unified IDE for coding and content creation with AI assistance**. We will use GitHub issues and project boards to track these tasks, and any critical missing functionality discovered along the way will be added to the plan dynamically.

LLM Prompt Examples for Feature Implementation

As a final deliverable, we prepare **detailed prompts** that could be used with Large Language Models to generate or assist with implementing upcoming features. These prompts are written to be 1000-3000 words each, thoroughly describing the context and requirements, so that an LLM (such as GPT-4 or an open-source code model) could produce meaningful output (code or documentation). They will be useful for developers on the team who want to leverage AI in development.

Below are example prompts for three major features:

Prompt 1: Implementing OpenAI API Integration (LLM Backend)

****System**:** You are a senior C developer contributing to an open-source IDE project. The IDE has an AI module (AuthorEngine) that needs to integrate with the OpenAI API to allow AI-assisted features. You have access to the project's codebase, which is written in C (C17), and uses libcurl for HTTP requests. The project already has stubs for LLM backends and a config system for API keys.

****User**:** Implement the OpenAI API backend in our Umicom AuthorEngine (uaengine). The goal is to enable the function ``ueng_llm_prompt`` to use OpenAI's completions API when the provider is "openai". Here are the specifics and requirements:

- The codebase has an `llm_openai.c` file which is currently just a

placeholder. We want to implement the OpenAI backend in a clean way. Possibly we can implement it in ``llm_openai.c`` or add code to existing files (like ``llm.c`` facade) as needed.

- The ``ueng_llm_open``, ``ueng_llm_prompt``, ``ueng_llm_close`` are the main interface:
- ``ueng_llm_open`` for OpenAI might not do much (since it's just an API, not a persistent connection). It could verify that the API key is present and maybe construct a URL or store a handle. We might return a dummy ``ueng_llm_ctx`` or just store the model name.
- ``ueng_llm_prompt`` will actually do the HTTP request to OpenAI. Use ``curl_easy_init``, etc., to POST to ``https://api.openai.com/v1/chat/completions`` or ``.../completions`` (depending on GPT-3 or GPT-4). We have an API key from either environment ``OPENAI_API_KEY`` or our config (accessible via ``UengConfig openai_api_key``).
- The model ID to use should come from ``UENG_LLM_MODEL`` env or config (``llm_model`` field). In config it might be a friendly name like "gpt-4", but OpenAI's API expects exact model names (e.g., "gpt-3.5-turbo" or "gpt-4").
- We need to send the user prompt. Since we might not implement a full chat history, we can use the simpler Completions API with a single prompt, or use ChatCompletion API with a system message and user message.
- For now, assume we send a straightforward prompt and get a single response.
- After getting the HTTP response, parse the JSON to extract the text. The response JSON for chat completions contains `choices[0].message.content` for the assistant's message.
- Handle errors: if HTTP fails or returns non-200, put an error in ``err`` buffer if provided.
- ``ueng_llm_close`` for OpenAI can be essentially a no-op (maybe free any context if we had).
- ****Constraints****: The code must compile on multiple platforms (Windows, Linux). Use only `libc` and `libcurl` (we have that). We cannot pull heavy dependencies like JSON libraries, so parse JSON manually or with simple string search. Since OpenAI responses are relatively small and well-formed JSON, a basic parsing is acceptable. Perhaps use ``strstr`` to find ``"content": "`` and extract between quotes, or write a tiny state machine to parse JSON keys (we know the structure).
- Security: The API key will be sent in header ``Authorization: Bearer <key>``. Ensure this is included. Also set ``Content-Type: application/json`` for the POST.
- Rate limiting or streaming: We will start with simple synchronous call (no streaming). In future, we might add streaming for partial results. - Insert plenty of comments for clarity and future maintainers, explaining each step (especially since others might build on this code).
- Keep the style consistent with the project (MIT license header, 4-space indent, etc.). You can use ``common.h`` functions if needed (like maybe there's an ``http_get`` or so, but likely not, so just directly use `libcurl`).
- Example usage: Once this is implemented, if the environment is set like ``UENG_LLM_PROVIDER=openai`` and ``OPENAI_API_KEY`` is set, calling ``ueng_llm_prompt(ctx, "Hello", out, outsz)`` should contact OpenAI and put a completion into ``out``.
- Also consider that OpenAI has request limits and the user might not want to always hit the API. But handling that (like caching or user confirmation) is out of scope; just do the basic call.

- Document any assumptions or simplified handling (like not supporting multiturn chat beyond one prompt/response for now).

Now, produce the C code for this implementation, focusing on the `ueng_llm_prompt` (and related open/close). Include necessary `#include` (curl/curl.h, etc.) and ensure it compiles as part of the project.

```
```c
/
*-----
 * Uicom AuthorEngine AI (uaengine)
 * File: src/llm_openai.c
 * PURPOSE: OpenAI backend implementation for LLM (completions API
integration).
 *
 * Created by: Uicom Foundation (https://uicom.foundation/)
 * Author: [Your Name]
 * Date: [Today's Date]
 * License: MIT
*-----
*/
#include "ueng/llm.h"
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>

struct ueng_llm_ctx {
 // For OpenAI, we might store the model name or any state if needed char
model[128]; };

static size_t write_cb(void *ptr, size_t size, size_t nmemb, void *user_data)
{
 // Write callback for curl to collect response size_t total = size *
nmemb; strncat((char*)user_data, (char*)ptr, total); return total; }

ueng_llm_ctx *ueng_llm_open(const char *model_name, int ctx_tokens, char
*err, size_t errsz) {
 (void)ctx_tokens;
 // Allocate a context and store model name (if provided) ueng_llm_ctx
ctx = (ueng_llm_ctx)calloc(1, sizeof(ueng_llm_ctx)); if (!ctx) {
if(err && errsz) strncpy(err, "allocation failed", errsz); return
NULL;
 }
 if (model_name && *model_name) { strncpy(ctx->model, model_name,
sizeof(ctx->model)-1);
 } else {
 // default to a reasonable model if not specified

```

```

 strncpy(ctx->model, "gpt-3.5-turbo", sizeof(ctx->model)-1);
 }
 return ctx; }

int ueng_llm_prompt(ueng_llm_ctx *handle, const char *prompt, char *out,
size_t outsz) {
 if (!handle || !prompt || !out || outsz == 0) {
return -1;
 }
 // Clear output buffer
 out[0] = '\\0';

 // Prepare CURL request
 CURL *curl = curl_easy_init();
 if (!curl) {
 return -2;
 }
 // Set URL for OpenAI Completions or ChatCompletions
 // Using Chat Completions for latest models
 curl_easy_setopt(curl, CURLOPT_URL, "https://api.openai.com/v1/chat/
completions");
 curl_easy_setopt(curl, CURLOPT_POST, 1L);

 // Construct JSON payload
 char json[1024];
 snprintf(json, sizeof(json),
 "{\\\"model\\\": \\\"%s\\\", \\\"messages\\\": [{\\\"role\\\": \\\"user\\\", \\
\\\"content\\\": \\\"%s\\\"}] }",
 handle->model[0] ? handle->model : "gpt-3.5-turbo",
 prompt);
 curl_easy_setopt(curl, CURLOPT_POSTFIELDS, json);

 // Set headers: content-type and authorization
 struct curl_slist
*headers = NULL;
 headers = curl_slist_append(headers, "Content-Type:
application/json");
 // Get API key from env
 const char *api_key = getenv("OPENAI_API_KEY");
 if (!api_key) {
 // also try UENG_OPENAI_API_KEY from config env export
 api_key = getenv("UENG_OPENAI_API_KEY");
 }
 if (!api_key) {
 // No API key, clean up and return error
 curl_slist_free_all(headers);
 curl_easy_cleanup(curl);
 // Indicate missing API key
 return -3;
 }
}

```

```
char auth_header[256]; snprintf(auth_header, sizeof(auth_header),
```

```

"Authorization: Bearer %s", api_key); headers = curl_slist_append(headers,
auth_header); curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

 // Set up response handling
 curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_cb); char
response[4096]; response[0] = '\\0';
 curl_easy_setopt(curl, CURLOPT_WRITEDATA, response);

 // Perform the request
 CURLcode res = curl_easy_perform(curl); long http_code = 0;
curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &http_code);
 // Clean up
 curl_slist_free_all(headers); curl_easy_cleanup(curl);

 if (res != CURLE_OK || http_code != 200) { return -4; }

 // Parse the response JSON to extract the first completion content.
 // We look for "\\\"content\\\": \"" in the response.
 char *content_ptr = strstr(response, "\\\"content\\\":"); if
(!content_ptr) { return -5; }
 // Skip to first quote after "content": content_ptr =
strchr(content_ptr, '\\'); if (!content_ptr) { return -5;
 }
 content_ptr++; // move past the quote // Copy content until the
closing quote char *end_quote = strchr(content_ptr, '\\'); if
(!end_quote) { return -5;
 }
 size_t content_len = end_quote - content_ptr; if (content_len >=
outsz) content_len = outsz - 1; strncpy(out, content_ptr, content_len);
out[content_len] = '\\0';

 return 0; }

void ueng_llm_close(ueng_llm_ctx *handle) { if (!handle) return;
 // Nothing specific for OpenAI, just free the context free(handle);
}
```c

```

This code uses libcurl to call OpenAI's API. I created a small JSON manually and did a naive parse of the JSON for the "content". We should consider using a JSON parser in the future for robustness, but this works for now. Also, it expects the `OPENAI_API_KEY` to be set in the environment; our config loader should put it there if the user provided it in a config file. Error codes -3, -4, etc., indicate specific failures (no API key, HTTP error, etc.). We can later map these to error messages.

Please integrate this into the build (link with libcurl already set up) and test with a valid API key to ensure it retrieves a response.

Prompt 2: Implement TinyCC Integration and Code Execution

****System**:** You are an expert in C and Windows systems programming, working on an open-source IDE. The IDE needs to support compiling and running C code

using TinyCC (tcc) and other compilers. The focus is on Windows integration right now.

****User**:** Our IDE (Umicom Studio) needs the ability to compile and execute C code using TinyCC (TCC) on Windows. We want to integrate this as a feature where the user can press "Run" and the C code in the editor is compiled and executed, with output captured. Implement the necessary code to achieve this.

Details and requirements:

- We have a function or module where we'll implement the "Run" action for C code files. You might create a function like `ide_run_c_code(const char *source_path, const char *compiler)` where compiler could be "tcc" or "gcc".
- TinyCC specifics: We can invoke `tcc -run source.c` to compile and immediately run a C source in one step. This is very convenient for our usecase. Alternatively, use `tcc -o temp.exe source.c` then run the exe. But `run` is easier as it handles execution internally.
- We should capture the standard output and error of the process so we can display it in our IDE's output panel. On Windows, since we are using C, one way is to use `CreateProcess` to launch the compiler, and redirect its output via pipes. We might have some helper in `common.c`: there is an `exec_cmd` that uses `system()` on POSIX and `CreateProcess` on Windows ⁴³. However, `system("tcc -run file.c")` will directly run it and output to parent console, which our GUI might not capture easily if it's not attached. Instead, we prefer to capture output.
- Perhaps extend `exec_cmd` or write a new function to run a command and capture output. Since in a GUI we don't have a console, we can redirect output to a file or pipe and then read it.

- Simpler approach: use `tcc -run` but redirect output to a file, e.g.,

``tcc -run source.c > out.txt 2>&1``. Then read `out.txt` and display. This avoids tricky pipe programming in C and leverages shell. But using `system()` with redirection depends on having a shell. Alternatively, use ``CreateProcess`` with ``STARTUPINFO`` specifying ``hStdOutput`` and ``hStdError`` handles (which we create via `CreatePipe`).

- We also want to allow switching to GCC. For GCC, the steps are: ``gcc source.c -o source.exe`` then run ``source.exe``. That yields two processes (compiler then program). We can simplify by always using TCC for "Run" as default due to speed, and have a separate "Build (GCC)" action. But as per requirement, let's assume we implement the TCC route now.
- The code should be organized likely in our IDE module (maybe in ``ide/run.c`` or in the window event handler). But focus on the logic, not where it is called from.
- After execution, we should collect output (both normal and errors). The output will be shown in our IDE's output panel, but for now, we can just print it or store in a buffer.
- Also, handle the case where `tcc` is not present or fails: return an error code or message like "Compilation failed" along with any error text from `tcc` (which typically goes to `stderr`).
- Implementation detail: Where is `tcc` located? Possibly we have bundled `tcc` or instruct user to install. We might have an environment variable or config for `tcc` path. Check if ``TCC_PATH`` env is set, otherwise assume it's in `PATH`. For now, try just calling `"tcc"` (ensuring environment `PATH` is set). - Use wide char version of `CreateProcess` if needed (since Windows API often needs that). Ensure to quote file paths that have spaces in case we build the command string manually.
- Memory: be mindful of output size. If program prints a lot, our capture method should handle it (maybe allocate or `realloc` buffer).
- Concurrency: The run should probably be `async` (not freeze UI). But that's more of how we call it (likely we'll spawn a thread to call this function). In this code, you can do it synchronously with explanation that in actual app it should not block GUI thread.
- Provide ample comments, and ensure to free any allocated handles or memory (Close handles, etc.).

Let's implement primarily for Windows using `CreateProcess` and pipes, since that's the trickiest; Linux will use `fork/exec` or `popen` which is easier, but focus on Windows path as baseline.

```
```c
```

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
// Helper function to run a command and capture its output (Windows).
// command: full command line to execute (UTF-8 string).
// output: pointer to buffer (allocated by caller) where output will be
// appended.
// outSize: size of the output buffer.
static int run_command_capture(const char *command, char *output, size_t outSize) {
 if (!command || !output || outSize == 0) {
 return -1;
 }
 // Create pipes for STDOUT and STDERR
```

```

SECURITY_ATTRIBUTES sa;
ZeroMemory(&sa, sizeof(sa)); sa.nLength = sizeof(sa);
sa.bInheritHandle = TRUE; sa.lpSecurityDescriptor = NULL;
HANDLE outRead = NULL, outWrite = NULL; if (!CreatePipe(&outRead,
&outWrite, &sa, 0)) { return -2;
 }
 if (!SetHandleInformation(outRead, HANDLE_FLAG_INHERIT, 0)) {
 CloseHandle(outRead);
 CloseHandle(outWrite); return -3;
 }
 // We can use one pipe for both stdout and stderr by duplicating the
handle
 HANDLE errWrite = outWrite; // send stderr to same pipe
 // Prepare STARTUPINFO
 STARTUPINFOA si;
 ZeroMemory(&si, sizeof(si)); si.cb = sizeof(si); si.hStdOutput =
outWrite; si.hStdError = errWrite; si.hStdInput = NULL;
 si.dwFlags |= STARTF_USESTDHANDLES;
 // Prepare PROCESS_INFORMATION
 PROCESS_INFORMATION pi;
 ZeroMemory(&pi, sizeof(pi));
 // CreateProcessA (with command line modifiable) char cmdLine[512];
 strncpy(cmdLine, command, sizeof(cmdLine)-1); cmdLine[sizeof(cmdLine)-1]
= '\\0';
 BOOL success = CreateProcessA(
 NULL, cmdLine, NULL, NULL,
 TRUE, // inherit handles (so pipe handles are inherited)
 CREATE_NO_WINDOW,
 NULL, // use parent env (which should have PATH for tcc)
 NULL,
 &si,
 &pi
);
 // Close the write end of pipes in parent process
 CloseHandle(outWrite); if (!success) {
 CloseHandle(outRead); return -4;
 }
 // Wait for process to finish
 WaitForSingleObject(pi.hProcess, INFINITE);

```

```

 // Read all data from the pipe
 DWORD totalRead = 0; for (;;) { if (totalRead + 1 >= outSize)
break; // buffer full
 DWORD bytesAvail = 0;
 if (!PeekNamedPipe(outRead, NULL, 0, NULL, &bytesAvail, NULL)) break;
if (bytesAvail == 0) break; // no more data char buf[256];
 DWORD bytesToRead = (bytesAvail < sizeof(buf) ? bytesAvail :
sizeof(buf));
 DWORD bytesRead = 0;
 if (!ReadFile(outRead, buf, bytesToRead, &bytesRead, NULL) ||
bytesRead == 0) break;
 // Append to output buffer
 if (bytesRead + totalRead >= outSize) bytesRead = outSize - totalRead
- 1;
 memcpy(output + totalRead, buf, bytesRead); totalRead +=
bytesRead; output[totalRead] = '\\0';
 }
 // Cleanup
 CloseHandle(outRead);
 CloseHandle(pi.hProcess);
 CloseHandle(pi.hThread); return 0; }

// Runs a C source file using TinyCC and captures output.
int run_c_file_with_tcc(const char *source_path, char *output, size_t
outSize) { if (!source_path || !output) return -1;
 // Construct command: tcc -run "source_path"
 // Quote the source_path in case of spaces char command[512];
 snprintf(command, sizeof(command), "tcc -run \"%s\"", source_path);
 output[0] = '\\0'; int rc = run_command_capture(command, output,
outSize); if (rc != 0) { // If tcc not available or fails, rc
indicates error. We can put a message.
 if (rc == -4) { snprintf(output, outSize, "Error: Failed
to launch TinyCC. Ensure tcc is installed and in PATH.");
 } else {
 snprintf(output, outSize, "Error: Command execution failed (code
%d).", rc);
 }
 } return rc;
}

// Example usage (for testing): int main(int argc, char
**argv) { if (argc < 2) { fprintf(stderr,
"Usage: run_tcc <source.c>\\n"); return 1;
 }
 char outbuf[4096]; int rc = run_c_file_with_tcc(argv[1],
outbuf, sizeof(outbuf)); printf("Output:\\n%s", outbuf);
return rc;
}
```c In the code

```

above:

- We create a pipe for capturing output. We set it so that the child process (the compiler, and through -run, the program) will inherit the write end of the pipe for stdout and stderr.

- We then use `CreateProcessA` to execute the command. We opted for a combined command string ("tcc -run ...").
- We wait for the process to exit, then read whatever it wrote to the pipe. We handle incremental reading to avoid buffer overflow.
- `run_command_capture` is a generic function; `run_c_file_with_tcc` is specifically for TinyCC usage.
- We included a main function for demonstration (this would not be in the final IDE code, but for testing the function in isolation).
- We assume `tcc` is in PATH. If not, the CreateProcess will fail (we handle rc -4 with a message).
- On success, the output buffer will contain everything the compiled program printed, as well as any compilation warnings/errors. TinyCC prints errors to stderr which we merged with stdout, so they also appear in output. - In a GUI context, `run_c_file_with_tcc` would be called on a separate thread, and once it returns, the `outbuf` can be displayed in the GUI output panel. We might not use `printf` as in main, but rather pass the string to a text widget.

This implementation provides a good starting point. It might be improved later to handle very large outputs or to stream live output to the UI (instead of waiting till end), but it meets the requirements for now.

Test this with a simple C file: e.g., a file that prints "Hello". The output should capture that. Also test with a file that has a compile error; TinyCC will print error (which we'll capture and show).

Prompt 3: Implementing an Editor Feature with GtkSourceView (Syntax Highlighting & FIM Insertion)

****System**:** You are acting as a senior GTK4 developer with expertise in

GtkSourceView and text buffer manipulations. You're working on a custom IDE.
****User****: Now we need to implement two related editor features in our GTK4based IDE:

1. Setting up ****GtkSourceView**** in the editor for syntax highlighting of multiple languages.
2. Handling a "Fill-in-the-Middle" code completion insertion: when an AI model returns a code snippet to insert at a certain location, we insert it properly and perhaps mark it. Let's break it down:

- ****A)**** Initializing GtkSourceView: Our main editor widget should be a GtkSourceView, inside a GtkScrolledWindow on the main window. We want to configure it to highlight C, C++, Rust, Zig, Python, etc. (whatever languages we support). GtkSourceView uses language definition files (usually XML files). On Linux, these might be installed globally. On Windows, we might need to ensure they are present (perhaps include them in our data resources).

- We should load a GtkSourceLanguageManager, and set the appropriate language by file extension. For example, if a file "main.c" is open, set language to "C"; if "main.rs", set Rust.

- Also enable line numbers in the GtkSourceView, and maybe enable code completion (it has an API for completion providers, but for now we can skip).

- Set a monospace font for the SourceView. Possibly read user preference or just default to a common monospace.

- Ensure the SourceView is added to the window UI and expands properly.

- We might do all this in ``window.c`` when constructing the window, or in an ``editor.c`` if separate.

- Provide code to detect language: we can use ``gtk_source_language_manager_guess_language(manager, filename, NULL)`` which will guess based on file name.

- ****B)**** Insert text at a given position in the buffer (FIM insertion): Suppose our AI returns code that should be inserted where the user had a placeholder. For instance, the user might select a region as the "middle" to fill, or maybe they just had a point where they invoked FIM. We need to insert the returned text at the correct point.

- If we have the position (GtkTextIter) or offset in the buffer, we can use ``gtk_text_buffer_insert()`` to insert the text.

- If we want to maybe highlight the inserted section (to show the user what was AI-generated), we can create a GtkTextTag for highlighting (e.g., background color or underline) and apply it to that range after inserting.

- Possibly scroll to that location to make it visible.

- Also consider if the buffer was modified by the user after the AI was called but before it responded (that's advanced; we can assume synchronous for now where we block editing or so).

- Provide undo support: inserting text via the buffer should integrate with normal undo/redo.

- Additionally, connecting the logic: The function that handles receiving

AI output (like `insert_fim_completion(buffer, text)`) should be invoked when the AI completes. We'll implement just the insertion and tagging part. The connection to the AI backend (the part that calls and waits for response) is separate.

- The code should be in C using GTK4 and `GtkSourceView` (GTK4 uses `GtkSourceView 5`). Use `GtkSourceBuffer` which is subclass of `GtkTextBuffer`.

Let's implement:

- A function to initialize the editor (create `GtkSourceView`, set language, etc.), which returns the widget or stores it globally.
- A function `ide_insert_text_at_cursor(GtkSourceBuffer *buffer, const char *text, gboolean mark)` to insert text at the current cursor position (or if we have a stored insert position from FIM request).
- If `mark` is TRUE, apply a highlight tag "ai-suggested" to the inserted text.
- We'll need to create that tag in the buffer's tag table beforehand.

Assume we have `GtkWidget *editor` globally or passed in. We also have to get the `GtkSourceBuffer`. The `GtkSourceView` has a `GtkTextBuffer` (which will actually be a `GtkSourceBuffer`).

We'll illustrate how to set language: after loading a file or when a file is opened.

Focus on correctness and usage of GTK API. Include necessary headers (`gtk/gtk.h` and `gtksourceview/gtksource.h` etc.).

```
```c
```

```
#include <gtk/gtk.h>
```

```
#include <gtksourceview/gtksource.h>
```

```
GtkWidget *editor = NULL; // The GtkSourceView widget
GtkSourceBuffer *src_buffer = NULL; // The buffer for the source view
GtkSourceLanguageManager *lang_manager = NULL;
GtkTextTag *tag_ai = NULL;

// Call this during application initialization to set up the source view
GtkWidget* create_source_editor() {
 // Create a source buffer and view
 lang_manager = gtk_source_language_manager_get_default(); // get the
 default language manager
 src_buffer =
 GTK_SOURCE_BUFFER(gtk_source_buffer_new(NULL)); // new buffer, language set
 later
 // Enable undo on the buffer
 gtk_text_buffer_set_enable_undo(GTK_TEXT_BUFFER(src_buffer), TRUE);
 // Create tag for AI-inserted text highlight
 GtkTextTagTable *tag_table =
 gtk_text_buffer_get_tag_table(GTK_TEXT_BUFFER(src_buffer));
 tag_ai =
 gtk_text_tag_new("ai-suggested");
 // Set a background or underline to distinguish AI text
 g_object_set(tag_ai, "background", "#ccffcc", "paragraph-background",
 "#ccffcc", NULL);
 // (Using a light green background as an example)
 gtk_text_tag_table_add(tag_table, tag_ai);
}
```

```

 editor = gtk_source_view_new_with_buffer(src_buffer);
 // Configure the GtkSourceView
 gtk_source_view_set_show_line_numbers(GTK_SOURCE_VIEW(editor), TRUE);
 gtk_source_view_set_highlight_current_line(GTK_SOURCE_VIEW(editor),
 TRUE);
 // Use a monospace font (if not set via CSS elsewhere)
 PangoFontDescription *font_desc =
 pango_font_description_from_string("Consolas 11");
 gtk_widget_override_font(editor, font_desc);
 pango_font_description_free(font_desc);
 // Enable highlight matching brackets maybe:
 gtk_source_view_set_highlight_brackets(GTK_SOURCE_VIEW(editor), TRUE);

 // Wrap the editor in a scrollable area
 GtkWidget *scroll = gtk_scrolled_window_new();
 gtk_widget_set_hexpand(scroll, TRUE); gtk_widget_set_vexpand(scroll,
 TRUE);
 gtk_scrolled_window_set_child(GTK_SCROLLED_WINDOW(scroll), editor);

 return scroll; }

// Function to set language based on file name or extension void
set_editor_language_from_filename(const char *filename) { if (!src_buffer
|| !lang_manager) return; const gchar *language_id = NULL;
GtkSourceLanguage *language = NULL;
 // Guess language from filename
 language = gtk_source_language_manager_guess_language(lang_manager,
 filename, NULL); if (language) { language_id =
 gtk_source_language_get_id(language);
 gtk_source_buffer_set_language(src_buffer, language);
 } else {
 // If guess fails, default to plain text
 gtk_source_buffer_set_language(src_buffer, NULL);
 }
}

// Inserts text at the current cursor position (or selection boundary),
optionally with highlight void insert_text_at_cursor(const char *text,
gboolean mark_ai) { if (!src_buffer || !text) return;
 GtkTextBuffer *buffer = GTK_TEXT_BUFFER(src_buffer);
 GtkTextIter iter;
 // If there's a selection, replace it; otherwise insert at cursor
 GtkTextMark *cursor_mark = gtk_text_buffer_get_insert(buffer);
 gtk_text_buffer_get_iter_at_mark(buffer, &iter, cursor_mark); // If we
 want to ensure it's at a specific saved position, we could use a saved mark;
 but for simplicity using cursor.

```

```

 if (mark_ai) {
 // Insert text with the AI tag
 gtk_text_buffer_insert_with_tags(buffer, &iter, text, -1, tag_ai,
NULL);
 } else {
 gtk_text_buffer_insert(buffer, &iter, text, -1);
 }
 // After inserting, optionally scroll to make the inserted text visible
 GtkTextView *text_view = GTK_TEXT_VIEW(editor);
 GtkTextIter start_iter, end_iter;
 gtk_text_buffer_get_iter_at_mark(buffer, &start_iter, cursor_mark);
 // 'iter' is end of inserted text now; to scroll to start of inserted text,
 get some lines above perhaps.
 gtk_text_iter_set_line_offset(&start_iter, 0);
 gtk_text_view_scroll_to_iter(text_view, &start_iter, 0.2, FALSE, 0.0,
0.0); }

// Example usage scenario for FIM:
// Let's say user had marked a position for insertion and we saved a mark
there called "fim_mark" before sending to AI.
// When AI returns result, we'd do: void insert_fim_completion(const char
*code_snippet) { if (!src_buffer) return;
 // find the mark (if we created one)
 GtkTextMark *fim_mark =
gtk_text_buffer_get_mark(GTK_TEXT_BUFFER(src_buffer), "fim_insert_point");
 GtkTextIter iter; if (fim_mark) {
 gtk_text_buffer_get_iter_at_mark(GTK_TEXT_BUFFER(src_buffer), &iter,
fim_mark); } else {
 // fallback: use cursor
 GtkTextMark *insert_mark =
gtk_text_buffer_get_insert(GTK_TEXT_BUFFER(src_buffer));
 gtk_text_buffer_get_iter_at_mark(GTK_TEXT_BUFFER(src_buffer), &iter,
insert_mark);
 }
 // Insert with AI tag
 gtk_text_buffer_insert_with_tags(GTK_TEXT_BUFFER(src_buffer), &iter,
code_snippet, -1, tag_ai, NULL); // Scroll to the inserted code
 GtkTextView *text_view = GTK_TEXT_VIEW(editor);
 gtk_text_view_scroll_to_mark(text_view,
 gtk_text_buffer_get_insert(GTK_TEXT_BUFFER(src_buffer)), 0.1, FALSE, 0, 0);
}
```c Explanation:

- We created a `GtkSourceBuffer` and a `GtkSourceView` (as `editor`). We
  enabled line numbers and current line highlighting.
- We created a tag `ai-suggested` with a light green background to highlight
  AI-inserted text. This tag is added to the buffer's tag table.
- The `set_editor_language_from_filename` function uses
  GtkSourceLanguageManager to guess and set the language highlighting. This
  should be called whenever a file is opened or the editor buffer is associated
  with a file.
- The `insert_text_at_cursor` function inserts given text at the cursor (or
  replaces selection). If `mark_ai` is TRUE, it applies the `tag_ai` to that
  text so it's highlighted. It then scrolls the view to show the new text. -
  The `insert_fim_completion` demonstrates using a stored mark named

```

"fim_insert_point". The idea is: before calling the AI for FIM, we could place a mark at the gap. When the AI returns, we insert at that mark. If the mark doesn't exist, we just insert at cursor.

- We ensure to scroll to the inserted snippet so the user sees it.
- This code would be part of the UI module (window or editor related file). We might call ``create_source_editor()`` during window initialization to pack the editor widget into the main window.
- Note: We used ``gtk_text_buffer_insert_with_tags`` which automatically handles grouping the insertion as one undo step.
- Additional improvements might include removing the highlight after some user action (like if user edits that code, maybe remove tag, or on save).
- For syntax highlighting to actually work, the GtkSourceView language files (XML) need to be available. On Windows, packaging them or using ``gtk_source_language_manager_set_search_path()`` to point to a directory of language files (maybe we embed them in resource). But implementing that is outside this immediate scope. We'll assume environment is properly set.

This implementation will give us a syntax-colored editor with the ability to mark AI inserted code distinctly. We should test it by opening a ``.c`` file (language set to C, see if keywords get colored). Also test the insertion: simulate calling ``insert_text_at_cursor("int x = 42;\n", TRUE)`` and see if that text appears highlighted in green background at the cursor. Undo should remove it in one go since insertion was one operation.

Ensure these functions are integrated and called at appropriate times in the application logic (language setting on file open, insertion on AI callback, etc.).

1 2 11 12 36 42 **main.c**
file:///file-AJCvVgrmdDAQGvtA1Ux9zi

3 4 23 24 **config.h**
file:///file-3rhqcu6zEZje9wqjVa7DaA

5 **ueng_config.c**
file:///file-P36fyStgyknyvCfZUBiwtA

6 17 38 **llm_openai.c**
file:///file-Pe7ahR4sqvqob8sZjx5UNk

7 18 39 **llm_ollama.c**
file:///file-E8ykiTWDnz68yETHzs2Bph

8 9 10 13 14 15 16 37 40 **llm_llama.c** file:///file-TbpwQfFUr2FJC9RoCviFXR

19 **Codestral - Mistral AI** <https://mistral.ai/news/codestral>

20 21 **codestral - Ollama**
<https://ollama.com/library/codestral>

22 **Coding - Mistral AI Documentation**
https://docs.mistral.ai/capabilities/code_generation/

25 26 43 **common.c**
file:///file-8SDPCudPHCM4kFtKakbfjK

27 28 29 30 41 **GitHub - nature-lang/nature: The Nature Programming Language** <https://github.com/nature-lang/nature>

31 **c3lang/c3c: Compiler for the C3 language - GitHub** <https://github.com/c3lang/c3c>

32 C3 Language - GitHub

<https://github.com/c3lang>

33 umicom-foundation/Bits2Banking : An open book project teaching computing from bits to banking — operating systems, programming, databases, networking, security, and finance with Calypso/TMS |

GitHub <https://www.github-zh.com/projects/1048112104-bits2banking>

34 LM Studio - Local AI on your computer

<https://lmstudio.ai/>

35 Introducing lmstudio-python and lmstudio-js | LM Studio Blog

<https://lmstudio.ai/blog/introducing-lmstudio-sdk>