ДЕТАЛЬНЫЙ ПЛАН РАЗРАБОТКИ WINDOWS-ПРИЛОЖЕНИЯ-АГЕНТА

Общее описание

---------------

Нужно разработать современное приложение-агент для Windows, которое будет управлять удалённым сервером LLM: устанавливать и проверять соединение, копировать скрипт `deploy\_llm\_server.py` на сервер и запускать его, отслеживать состояние сервера. Интерфейс должен быть модульным, плавным, без зависаний; поддержка многоядерности достигается за счёт использования отдельных процессов/потоков для тяжёлых операций. Приложение должно собираться в EXE и иметь простой режим отладки (вывод расширенных логов, доступ к Developer Tools).

Структура проекта

-----------------

Проект делится на две основные части: backend (службы и задачи) и frontend (пользовательский интерфейс).

1. \*\*Backend (каталог `electron-main` или `backend`)\*\*

\* `main.ts/js` — точка входа для Electron или иного фреймворка. Создаёт окно приложения, настраивает IPC.

\* `services/logger.ts` — асинхронный логгер: пишет логи в файл, добавляет временные метки, передаёт события логирования в UI.

\* `services/config.ts` — чтение и запись конфигураций: `app.yaml` (настройки приложения) и `servers.json` (список серверов). Предоставляет методы `getAppConfig()`, `getServers()`, `updateServer(server)`.

\* `services/sshService.ts` — работа с SSH/SFTP: метод `checkConnection(conf)` для проверки доступности, `copyFile(conf, local, remote)` для копирования файлов, `runCommand(conf, cmd)` для удалённого запуска команд, `setupTunnel(conf)` для создания SSH-туннеля.

\* `services/llmService.ts` — HTTP-клиент: `healthCheck(url)`, `getModels(url)`, `chat(url, messages)` с тайм‑аутами, читаемыми из конфигурации.

\* `services/taskExecutor.ts` — менеджер запуска задач в отдельных worker-потоках или процессах. Имеет методы `runTask(name, args)`, `cancelTask(id)`. Работает с API `worker\_threads` (Node) или `System.Threading` (.NET).

\* `services/serverManager.ts` — верхний уровень: предоставляет функции `testConnection(serverId)`, `deployServer(serverId)`, `connectServer(serverId)`, `ensureLLMReady(serverId)`. Использует `sshService`, `llmService` и `taskExecutor`. Хранит состояние серверов, отправляет прогресс/статус в UI.

2. \*\*Frontend (каталог `renderer` или `UI`)\*\*

\* `App.tsx` — корневой компонент, содержит маршрутизацию или главный макет.

\* `components/Chat.tsx` — отображает историю сообщений, поле ввода, кнопку отправки. По клику вызывает `llmService.chat()` через IPC.

\* `components/CodeEditor.tsx` — интегрирует Monaco Editor (web) или AvalonEdit (WPF), позволяет редактировать скрипты и конфигурации.

\* `components/DiffViewer.tsx` — использует библиотеку diff2html (web) или аналог для WPF для отображения различий.

\* `components/FileExplorer.tsx` — показывает структуру файлового проекта, поддерживает операции чтения и открытия.

\* `components/ServersPanel.tsx` — форма с полями IP, порт, SSH-ключ, путь проекта; кнопки “Test”, “Deploy”, “Connect”; статус, прогресс-бар и лог.

\* `components/SettingsDialog.tsx` — модальное окно для изменения настроек приложения (директории, тайм-ауты, языковые опции).

\* Хуки `useIpc` или сервисы для общения с backend через IPC.

3. \*\*Tasks (каталог `tasks`)\*\* — модули, которые выполняют тяжёлые операции в отдельных процессах/потоках:

\* `deployTask.ts` — принимает конфигурацию сервера, подключается по SSH, копирует `deploy\_llm\_server.py`, запускает его и следит за выходом. Отправляет сообщения в виде JSONL: тип события (`progress`, `log`, `complete`, `error`), значение прогресса, текст лога.

\* `connectTask.ts` — проверяет, что нужный порт свободен; запускает команду для создания туннеля; проверяет `/health` эндпоинт; отправляет прогресс.

\* `chatTask.ts` (опционально) — чтобы разгрузить основной поток, может выполнять запросы к LLM отдельно.

4. \*\*Tools\*\* — папка, куда помещается скрипт `deploy\_llm\_server.py` и другие вспомогательные скрипты, которые нужно передавать на сервер.

5. \*\*Конфигурации\*\* — `configs/app.yaml` (настройки интерфейса, тайм-ауты, режимы отображения), `config/servers.json` (массив объектов с полями: name, host, port, user, sshKey, deployPath, llmPort, status, deployed, connected, проектные пути).

Этапы разработки

----------------

### Этап 1. Настройка проекта

1. Создать пустой репозиторий и добавить файлы `.gitignore`, `README.md`.

2. Инициализировать проект:

\* для Node/Electron — `npm init -y`, установить `electron`, `react`, `react-dom`, `@types/react`, `typescript`, `ssh2`, `worker\_threads`, `electron-builder`;

\* для .NET/WPF — создать `WPF Application` в Visual Studio, подключить библиотеку `SSH.NET`, настроить `NuGet` для `Newtonsoft.Json`.

3. Настроить `electron-builder` или publish-профиль .NET для сборки exe.

4. Создать базовую структуру каталогов (`electron-main`, `renderer`, `tasks`, `tools`, `configs`).

5. Скопировать `deploy\_llm\_server.py` в папку `tools`.

### Этап 2. Реализация backend-служб

1. \*\*Логгер\*\*: создать класс, который пишет строки в файл (например, `logs/app.log`), ограничивает размер логов, делает ротацию. Добавить метод `log(level, message)` и событие/IPC `log` для отправки в UI.

2. \*\*Конфигурация\*\*: загрузить `app.yaml` и `servers.json` при старте, обеспечить методы чтения и обновления. В случае отсутствия файлов — создавать их с дефолтными значениями.

3. \*\*SSH-сервис\*\*:

\* Использовать `ssh2.Client` (Node) или `SshClient` (.NET) для подключения.

\* Реализовать `checkConnection(config)`: попытка соединения с тайм‑аутом.

\* Реализовать `copyFile(config, local, remote)`: использовать SFTP для передачи файлов.

\* Реализовать `runCommand(config, command)`: исполнить команду на сервере, возвращая stdout/ stderr.

\* Реализовать `setupTunnel(config)`: поднять SSH-туннель с пробросом порта `llmPort`.

4. \*\*LLM-сервис\*\*:

\* Функция `healthCheck(url)` возвращает `true`/`false`.

\* `getModels(url)` извлекает список моделей.

\* `chat(url, messages)` отправляет сообщение и возвращает ответ.

5. \*\*Task Executor\*\*:

\* Метод `runTask(name, args)`: создаёт worker‑поток (для Node — `new Worker()` с соответствующим скриптом), подписывается на сообщения `message` и `error`.

\* Метод `cancelTask(id)`: завершает worker.

\* Передаёт события `progress`, `log`, `complete`, `error` через IPC.

6. \*\*Server Manager\*\*:

\* Хранит список серверов из `servers.json` и активный сервер.

\* `testConnection(serverId)`: проверяет SSH и доступность LLM (вызов `taskExecutor` с `connectTask`).

\* `deployServer(serverId)`: выполняет копирование и запуск LLM (вызов `deployTask`).

\* `connectServer(serverId)`: запускает SSH-туннель и проверяет здоровье.

\* `ensureLLMReady(serverId)`: если сервер не развёрнут, вызывает `deployServer`, затем `connectServer`.

\* Обновляет поля `deployed`, `connected`, `status` в `servers.json` и отправляет события в UI.

### Этап 3. Реализация UI

1. Выбрать фреймворк:

\* Web: React + Tailwind/Material UI; использовать `electron-react-boilerplate` для быстрой интеграции.

\* Desktop: WPF + MVVM; использовать библиотеку FluentWPF для современного вида.

2. \*\*Главный макет\*\*:

\* Левая панель — список/форма серверов (ServersPanel);

\* Центральная часть — вкладки для редактора кода (CodeEditor), дифф (DiffViewer) и лог (FileExplorer/LogViewer);

\* Правая часть — чат (Chat).

3. Перенести стиль и расположение из прототипов: тёмная тема, закруглённые элементы, отступы.

4. Каждый компонент подключается через события и вызывает методы `Server Manager` или `LLM Service` через IPC.

5. Отображать прогресс и логи в реальном времени: подписка на события `progress`, `log`, `complete`.

6. Настройки (SettingsDialog) позволяют изменить тайм‑ауты, пути для проектов, язык интерфейса.

### Этап 4. Реализация задач (Tasks)

1. \*\*deployTask\*\*:

\* Получает конфигурацию сервера: адрес, порт, логин, путь для копирования, имя файла.

\* Использует SSH для соединения. Проверяет свободное место и установленный Python.

\* Копирует `deploy\_llm\_server.py` в указанную папку (`deployPath`) через SFTP.

\* Запускает `python3 deploy\_llm\_server.py`, читает вывод построчно, преобразует в события JSON (тип: progress, log).

\* По завершению возвращает `ok=true/false`.

2. \*\*connectTask\*\*:

\* Проверяет доступность порта `llmPort` локально.

\* Создаёт SSH-туннель: `ssh -L localPort:localhost:remotePort` (или делает это через API SSH библиотеки).

\* Отправляет периодические проверки `/health` до получения состояния `healthy`.

3. \*\*chatTask\*\* (если нагрузка на чат высока):

\* Получает список сообщений, вызывает `llmService.chat()`, возвращает результат.

\* Может использовать собственный тайм‑аут и отдавать промежуточный прогресс (например, счётчик попыток).

### Этап 5. Интеграция

1. Связать компоненты UI с сервисами backend через IPC:

\* Нажатие кнопки “Test” на ServersPanel посылает `testConnection` в main‑процесс и отображает прогресс.

\* “Deploy” посылает `deployServer`, “Connect” — `connectServer`.

\* Чат отправляет сообщение в LLM и отображает ответ.

2. Обеспечить обновление конфигурации после успешного развёртывания/подключения: установить `deployed=true`, `connected=true`, сохранить в `servers.json`.

3. Реализовать обработку ошибок: выводить дружелюбные сообщения пользователю и подробные логи в консоль.

4. При закрытии приложения аккуратно завершать все worker‑ы и SSH‑туннели, сохранять текущие конфигурации.

### Этап 6. Тестирование и отладка

1. \*\*Юнит-тесты\*\*: написать тесты для Logger, Config, SSH Service (имитировать SSH сервер), LLM Service (мок endpoints), Task Executor (обработка сообщений).

2. \*\*Интеграционные тесты\*\*: поднять тестовый SSH сервер (например, локальный контейнер), проверить развертывание `deploy\_llm\_server.py` и соединение.

3. \*\*UI-тесты\*\*: с помощью Spectron или Playwright протестировать цепочки действий пользователя.

4. \*\*Режим отладки\*\*: добавить параметр `debug=true` в config; если он включён — выводить Developer Tools (`BrowserWindow.webContents.openDevTools()` в Electron) и писать расширенные логи.

### Этап 7. Сборка и доставка

1. Настроить `electron-builder` (или publish-профиль .NET) для генерации `Setup.exe`/`App.exe`. Убедиться, что в пакет входит папка `tools` с `deploy\_llm\_server.py`.

2. Настроить `extraResources` для конфигураций и скриптов.

3. Тестировать установщик на «чистом» Windows: проверять, что приложение запускается, создаёт/читает конфиг, корректно отображает интерфейс.

4. (По желанию) настроить автоматические обновления (например, с использованием GitHub Releases).

Заключение

----------

Следуя этому плану, агент сможет последовательно разработать новый модульный клиент для работы с удалённым LLM-сервером. Главные принципы: разделение UI и тяжёлых операций, использование многопоточности/многопроцессности, чёткая структура каталогов, единый формат обмена сообщениями (JSONL) между задачами и интерфейсом, и тщательное тестирование на каждом этапе.