Представь, что я - домохозяйка, не владеющая технологиями разработки веб-сайтов и приложений. Я хочу написать (которую пользователь мог бы запускать с локального компьютера) со следующим функционалом.

Посмотри приложенный файл с экономическим исследованием - от чего зависит рост зарплат в стране.

Само исследование неплохое, но мне надо улучшать модель. Добавлять регрессоры, тестировать, прогонять заново и заново. Мне нужен сайт, который бы максимально упростил для пользователя "пересборку" и тесты моделей.

Алгоритм, исполненный в файле wagesmodel надо превратить в сайт с пайплайном модельной работы, разбить всю деятельность на блоки, достаточно независимые друг от друга.

Я представляю себе это так. Весь экран всегда поделен на две половины. Слева - элементы управления, справа - визуализация.

Блок №1

--- ЛЕвая часть
Импорт данных. Пользователь кидает эксель с данными в большую . Программа сама определяет - есть ли там хоть какие то time series, если нет, то выводит ошибку.
Если timeseries есть - программа автоматически вычленяет оттуда всю мета-дату, перечисляет пользователю ряды в виде интерактивных элементов-пластиночек.
Timeseries могут идти по строкам, могут по столбцам.

Кажый элемент

- Кликабелен - по нажатию происходит визуализация (в правой части - см далее)
- При наведении прям на элементе появляется кнопка с галочкой - по ее нажатию ряд будет отправляться на следующий этап. Элементы, который пользователь не выбрал - в дальнейшем анализе не участвуют.
- На каждом элементе выведен детектированная периодичность данного ряда.
- Если в файле для каждого ряда обнаружились другие метаданные - их надо вывести на этом же элементике.
  Например, может быть информация о размерности ряда, сущности - flow или level и тд. При наведении мышки на каждый маркер мета-данных - всплывает хинт - что это за тип мета-данных распознался.

В этом блоке программа сама понимает, какая периодичность данных для каждого ряда - могут быть ежедневные, еженедельные, ежемесячные ряды, квартальные, годовые и тд. Пользователю ниже предлагается выбор - к какому тайм фрейму все привести. Если он выбирает, допустим, Q - все ряды приводятся к этому тайм фрейму.

В интерфейсе импортера можно будет указать

1. Выпадающие списки с методами усреднения (с более мелких таймфреймов на более высокие, например, с M на Q) /экстраполяции (с более высоких тайм фремов на более мекие, например, с Q на M).

2. Выбор - на каких листах загруженного экселя искать данные.
   --- Правая часть - по нажатию на любой обнаруженный ряд данных - он отрисовывается на правой стороне, с возможностью приближения/удаления и тд

Блок №2:
--- Левая часть
Когда пользователь выбрал нужные ему ряды, появляется возможность пополнения, обогащения и всяческого улучшения датасета.

- Можно изменить имя ряда
- Можно дифференцировать ряд (создается новый синтетический) - либо в абсолюте, либо в виде % приращений. По таймфреймам тоже разные возможности - либо м/м, либо q/q, либо y/y
  Надо пользователю давать выбрать опции дифференциации
- Можно нормировать данный ряд на другой из отобранных.
  ... потом еще придумаем опции

Когда датасет очищен

--- Правая часть - аналогично как в прошлом блоке - визуализация любого ряда из обогащенного множества.

Блок №3 - модель
--- Левая часть
В самом верху блока - выбираем тип моделирования. Для начала реализуем для типа моделей "Регрессия".
Строить ОДНУ регрессию - нет никакого смысла. Имеет смысл только их перебрать сразу ВСЕ на данно множестве факторов. Поэтому перед пользователям опять возникает набор планочек с названиями факторов из обогащенного списка, все они отмечены как активные (т.е. регрессия будет искаться на множестве их подмножеств).

Внешний вид планочек почти такой же, как в первых двух списках, но не совсем. На планочках уже нет метаданных, периодичности и размерности, есть только название и статус (один из нескольких):

- Участвует в переборе (один цвет)
- Не участвует в переборе
- Обязательно должен присутствовать в топовой регрессии (что такое топовая регрессия - ниже).
- Опционально - задать знак и/или магнитуду коэффициента, с которым данный фактор должен входить в регрессию. При переборе регрессий спецификации, в которых знак коэффициента не совпадает с заданным, не будут претендовать на звание топовой.

Чуть ниже есть отдельный индикатор для константы в модели - она имеет один из трех статусов - "обязательно включать", "обзяательно не включать", "проверить на включение". В третьем случае она тестируется в общем множестве факторов наравне с другими потенциальными регрессорами.

Чуть ниже пользователь задает глубину лагов N - по сколько лагов проверить для каждой из переменных. Проверяется все лаги от 0 до N включительно. (рядом должно стоять

напоминание - на каком таймфрейме пользователь решил исследовать задачу - D, W, M, Q, Y, ...)

Чуть ниже пользователь выбирает, какие статистические тесты делать ему при переборе моделей - коллинеарность, гетероскедастичность, max pvalue (с задаваемой пользователем порогом), какие то еще тесты, которые ученые-эконометристы делают.

Чуть ниже пользователь отмечает, какие типы метрик точности алгоритм должен для него считать - r^2, mae, MAPE, RMSE, ...

Чуть ниже пользователю красиво большими числами выводится - всего будет осуществлен перебор X регрессий, где X - вычисляется (см мой код) из числа факторов и их статуса, а также глубины лагов.

Дальше стоит большая и красивая кнопка - "запустить перебор", пользователь ее нажимает, появляется визуализация перебора.

После запуска появляют Кнопки "пауза", кнопка "стоп".

Внимание. При переборе всех специфицкаций - мы их записываем в некий json файл (включая оценки коэфф, статистические метрики), по мере обработки. То есть, если вдруг пользователь решит нажать на паузу, потом выйдет из приложения, потом опять зайдет - чтобы прогресс перебора регрессий - не сбросился.

Прогресс может сброситься только в одном случае. Если пользователь сам открутил интерфейс к предыдущему блоку - и поменял в нем набор факторов, которые пойдут в "модельный" блок №3. В этом случае, если входной файл не был заменен - то мы просто сохраняем "слепки" блока №2 и блока №3 - как параллельной реальности, они становятся серыми, но к ним у пользователя будет возможность вернуться и продолжить исследовать. Но если человек, все таки, поменял что-то в блоке №2 или блоке №1 (но без замены исходного файла эксель), то создается "новая реальность", где блок №2 и блок №3 конструируются заново. Пользователь может переключаться между реальностями. И давай все таки лучше сохранять прогресс перебора для каждой реальности.

Если пользователь просто меняет ограничения на знаки, или делает некоторые из них обязательными - прогресс не сбрасывается, т.к. в новой постановке задачи регрессии то будут перебираться те же самые. ПРосто из множества перебранных регрессий некоторая часть перестанет быть валидными, а некоторая часть - наоборот начнет быть валидными.

Если пользователь включает/выключает переменные вверху блок №3 - то прогресс не сбрасывается, но происходят изменения. Те регрессии, которые были просеяны - их результаты остаются. Но общее множетсво регрессий либо увеличивается, либо уменьшается. Если пользователь выключает какую то переменную - то множество регрессий сужается. И среди него уже будут те специфицкации, которые обработчик уже перебрал. Если пользователь включает переменную - то множество регрессий для перебора расширяется, и все новые добавленные однозначно еще будут иметь статус "пока не исследована".

--- Правая часть.
Нам надо визуализировать процесс перебора. Только делать это наглядно, а не банальной процентовкой "мы перебрали x% из 100%". Давай визуализируем в правой части множество регрессий как набор квадратиков. Делать его просто плоской решеткой N*M

или как то более интересно - как будто на гиперкубе - подумай сама, главное, чтобы было красиво и наглядно.
Давай закодируем цвета:
Зеленый - регрессия проверена и валидна по заданным пользователям условиям.
Желтый - регрессия проверена, проходит статистические тесты - коллинеарность, гетероскедастичность и тд, но не проходит фильтр на знаки и магнитуды коэффициентов.
Красный - регрессия проверена, но не проходит статистические тесты.
Серый - регрессия еще не проверена.
Прозрачные с черным ободком - регрессии проверялись ранее, но сейчас вне контекста.

При изменении постановки - с квадратиками, в зависимости от ситуаций, перечисленных выше - происзодит следующее. Либо множество квадратиков прирастает (в случае с добавлением переменной к множеству перебора, увеличения числа лагов и тд), либо множество квадратиков сужается (при уменьшении числа лагов,), но уже перебранные регрессии не затираются, а перекрашиваются в статус прозрачных "вне контекста".

Каждый квадратик не рандомно отрисовывается, а в соответствии со спецификацией - на него можно нажать и чуть ниже в правой части экрана будет выведена вся информация о модели - регрессоры, точность, значения тестов и тд.

Если пользовтаель меняет специфицкацию, но мы понимаем, что модели полностью заново перебирать не надо - то мы лишь очень быстро проходимся по уже сохраненному json файлу с моделями - и перекрашиваем квадратики с точки зрения обновленных правил валидности.

Когда перебор моделей закончен - ни остается ни одного серого квадратика.

В ходе разработки в визуализацию третьего блока были внесены изменения, теперь он должен выглядеть так:

- **Верхняя часть:** Отображается сводная статистика по задаче (JobSummaryStats: статус, прогресс, кол-во валидных/невалидных и т.д.).
- **Фильтры:** Есть опция "Show Only Valid Models".
- **Основная визуализация:** Представлена в виде **Scatter Plot** (ModelScatterPlot), где модели отображаются как точки.
  - Оси X и Y можно выбирать из списка доступных метрик (R^2, MAE, Num. Regressors и т.д.).
  - Точки раскрашены по статусу: **зеленые (valid)**, **красные (invalid stats)**. Выбранная точка подсвечивается оранжевой звездой.
  - График не скроллится, занимает фиксированную часть экрана (примерно квадратный).
- **Панель деталей:** Появляется под Scatter Plot.
  - Отображает детальную информацию (в формате JSON) о модели, выбранной кликом на Scatter Plot или в таблице.
  - Панель можно свернуть/развернуть с помощью кнопки.
  - Изначально панель свернута и показывает только заголовок "Model Details", пока модель не выбрана.
- **Таблица результатов:** Отображается под панелью деталей (ModelResultsTable).
  - Содержит список завершенных моделей с основными метриками и результатами тестов.

    - Поддерживает сортировку по колонкам, глобальный поиск, пагинацию (30 строк по умолчанию).
    - Заголовок таблицы "залипает" при вертикальной прокрутке *внутри* таблицы.

- **Прокрутка:** Вся правая часть экрана (точнее, контейнер .visualization-content-wrapper внутри .right-pane) имеет вертикальную прокрутку, если все содержимое (ModelDashboard) не помещается по высоте.

Блок №4 - model viewer
--- Левая часть
Когда все регрессии перебрались - пользователь должен иметь возможность посмотреть результаты.
В самом начале блока (уже не в правой, а в левой части) копируется множество квадратиков, но только окрас уже другой. Активными остаются только те квадратики, которые имели цвет "зеленый" на предыдущем шаге.
Остальные все - становятся прозрачными с черным ободком.
Активные же приобретают окрас по типу heat map. Чем меньше ошибка по одной из выбираемой пользователем метрики (r^2, RMSE, MAE, ...) - тем теплее цвет. И наоборот, чем больше ошибка - тем холоднее цвет.

Пользователь опять имеет возможность нажать на квадрат, под ним так же выведется информация о модели, но в правой части еще и отрисуется график с декомпозицией.

--- Правая часть - выводит визуализацию регрессии - bar chart (как в моем приложенном коде, только с возможностью наведения мышки и появления подсказки со значениями факторов и вкладами их в модельный ряд в этой точке)

В самом низу появляются кнопки - "Сохранить все результаты" - всю информацию о топовой модели (лучшей с тз метрик точности) выводим в файл эксель.

В ходе решили, что четвертый блок делать не надо, а барчарт с визуализацией модели сделали прямо в правой части третьего блока. Нажимаешь на любую модель из таблицы – она сразу же отрисовывается.

Сайт должен быть максимально современный, динамичный, с привычной интеракцией для большинства людей!

Я бы предложил следующее при разработке придерживаемся следующих принципов:

1. Периодически сохраняем проект на гитхабе
2. Сама страница (сайт) должна хоститься с github pages
3. При разработке используем технологии react, node.js

Осознай план - все ли понятно тебе?

Here's a breakdown of the thinking process to generate the project plan for the interactive econometrics tool:

1. **Deconstruct the Request:**
    - **User Persona:** "Domokhozyaika" (housewife), non-technical, wants a *local* application (contradicts GitHub Pages later, need to address). Wants to simplify repetitive model testing.
    - **Core Problem:** Existing Python script (WagesModel.py) performs a complex econometric analysis (loading data, processing, variable selection, regression modeling, evaluation). This process is manual, requires code changes for iterations, and lacks interactivity.
    - **Desired Solution:** A web-like application (even if local) that transforms the script's logic into an interactive, visual pipeline.
    - **Key Features (by Block):**
        - **Block 1 (Data Import):**
            - Drag-and-drop Excel upload.
            - Automatic time series detection (rows/columns). Error if none.
            - Metadata extraction.
            - Interactive "plaques" for each series (visualize on click, select for next step, show metadata/periodicity).
            - Automatic periodicity detection (D, W, M, Q, Y).
            - Target timeframe selection (e.g., Q).
            - Resampling/interpolation method selection.
            - Sheet selection.
            - Right pane: Time series visualization (zoom/pan).
        - **Block 2 (Data Enrichment):**
            - Rename series.
            - Differentiate series (abs/%, m/m, q/q, y/y), create new synthetic series.
            - Normalize series by another.
            - Future options.
            - Right pane: Visualization of enriched series.
        - **Block 3 (Modeling - Regression):**
            - Select model type (initially just Regression).
            - Factor selection "plaques" with statuses:
                - In selection (default)
                - Not in selection
                - Required in *best* model
                - Optional: Sign/magnitude constraints (filter *best* model candidates).
            - Constant status (Include/Exclude/Test).
            - Lag depth selection (0 to N). Show target timeframe.
            - Statistical test selection (VIF, Heteroskedasticity, p-value threshold).
            - Accuracy metric selection (R^2, MAE, MAPE, RMSE).
            - Display estimated number of regressions to run.
            - "Run" button.
            - Pause/Stop buttons.
            - **Persistence:** Save regression results (coefficients, metrics, tests) incrementally to a JSON file. Resume on restart.
            - **"Realities" / Versioning:** If Block 1/2 inputs change (except file upload), create a new "reality" (snapshot) of Blocks 2 & 3. Allow switching between realities. Progress saved per reality. Changing constraints in Block 3 *doesn't* reset progress, just re-evaluates validity. Adding/removing variables *doesn't* reset progress but

expands/contracts the *search space*, potentially marking some regressions as "out of context".
        - **Right pane (During Run):** Visualize regression space (grid/hypercube). Colors: Green (valid), Yellow (stat ok, constraint fail), Red (stat fail), Grey (not run), Transparent/Outline (out of context). Clickable squares show model details below. Re-coloring happens quickly when constraints change.
    - **Block 4 (Model Viewer):**
        - **Left pane:** Copy of regression space visual from Block 3, but only showing Green squares from the previous step. Others are transparent. Active squares colored by heatmap based on a selected metric (R^2, MAE, etc. - warmer = better). Clickable squares show details below.
        - **Right pane:** Bar chart decomposition (like the Python script) for the selected model. Hover tooltips show values/contributions.
        - "Save Results" button (exports best model info to Excel).
  - **Technical Stack Suggestion:** React, Node.js, host on GitHub Pages.
  - **Design Principles:** Modern, dynamic, intuitive interaction. Periodic GitHub saves.
2. **Synthesize and Structure the Plan:**
  - **Goal:** Create an interactive, local-first (initially, reconcile with GH Pages later) web application for iterative econometric modeling based on the provided Python script's logic.
  - **Target User:** Non-technical researcher needing a visual and simplified workflow.
  - **Core Concept:** Break the Python script's workflow into distinct, user-controlled blocks (Import -> Enrich -> Model -> View) with a persistent state and visual feedback.
3. **Flesh out Each Block (Adding Detail and Technical Considerations):**
  - **Block 1 (Data Import):**
    - *Frontend (React):* File input (drag/drop), sheet selector, UI for resampling methods, display of detected series plaques, interactivity (click, hover, select). Visualization component (Plotly.js, Chart.js, etc.).
    - *Backend (Node.js/Python?):* Need logic to read Excel (xlsx library in Node, or potentially call a Python backend for pandas). Robust time series detection (look for date columns/rows, numerical data). Metadata parsing (needs clear rules based on example file). Periodicity detection (pandas infer_freq or similar logic). Resampling/interpolation logic (again, pandas is strong here, might need Python backend or port logic to JS).
    - *State Management:* Store uploaded data, detected series, metadata, user selections.
  - **Block 2 (Data Enrichment):**
    - *Frontend:* UI for renaming, selecting series for differentiation/normalization, specifying parameters (lag, %, base series). Display updated list of series. Visualization component.
    - *Backend/Logic:* Implement differentiation (diff, pct_change), normalization. Store transformations applied.
    - *State Management:* Store the enriched dataset and transformation history.
  - **Block 3 (Modeling):**
    - *Frontend:* UI for selecting factors, setting statuses (include/exclude/required/constraints), constant options, lag depth, test/metric selection. Display estimated run count. Run/Pause/Stop

buttons. Visualization of regression space (maybe use grid, SVG, or Canvas). Component to display selected model details.
        - *Backend/Logic (Crucial):* This is where the core statsmodels logic lives.
            - **Challenge:** statsmodels is Python. Options:
                1. **Full Python Backend:** Node.js acts as an API gateway, calling a Flask/Django/FastAPI Python service that runs the statsmodels code. This is the most straightforward way to reuse the existing logic.
                2. **WebAssembly (WASM):** Compile Python/statsmodels to WASM (e.g., Pyodide). Runs *in the browser*. Can be complex, might have limitations. Less suitable for long-running background tasks.
                3. **Port Logic to JS:** Rewrite statsmodels OLS, VIF, tests in JavaScript. Very difficult and error-prone. *Not recommended.*
            - **Decision:** Assume a Python backend is necessary for statsmodels. Node.js handles the UI logic and communicates with Python.
            - **Regression Runner:** Implement the iteration logic (combinations, lags). Needs to run potentially long tasks, possibly in the background (web workers if WASM, separate process/queue if Python backend).
            - **Persistence:** Design JSON structure for saving model results. Implement saving/loading logic. Handle Pause/Resume state.
            - **"Realities":** Implement state snapshotting and switching logic. Manage the "out of context" status based on changes.
            - **Re-evaluation:** Logic to quickly filter saved JSON results based on changed constraints.
        - *State Management:* Track selected factors, constraints, run progress, saved model results (JSON path/content), current "reality".
    - **Block 4 (Model Viewer):**
        - *Frontend:* Display filtered regression space (heatmap). Component for model details. Bar chart decomposition visualization (reuse charting library). "Save" button triggers export.
        - *Backend/Logic:* Filter the saved JSON based on validity from Block 3. Select the best model based on the chosen metric. Generate Excel export (using Python backend likely easiest).
        - *State Management:* Selected metric for heatmap, selected model for detail view.
4. **Address Technical Choices and Contradictions:**
    - **Local App vs. GitHub Pages:** The user wants a local app, but suggests GH Pages. GH Pages is for *static* sites. It cannot run a Node.js or Python backend.
        - **Resolution:** Propose two paths:
            1. **Electron App:** Bundle Node.js + React (+ potentially Python runtime) into a desktop application. Truly local.
            2. **Local Server:** User runs a local Node.js/Python server and accesses the app via localhost in their browser. Simpler than Electron, still local.
            - **Clarify:** Explain *why* GH Pages isn't suitable for the *backend* part and suggest alternatives. Stick with the local server model for the plan unless Electron is explicitly preferred.
    - **React/Node.js:** Good choices for the frontend and API layer.

    - **Python Backend:** Seems necessary for statsmodels. Need clear communication between Node.js and Python (e.g., REST API).
    - **Persistence:** Local file storage (JSON) is feasible for a local app/server setup. Need robust error handling for file I/O.
    - **Visualization:** Choose a library (Plotly.js is powerful, Chart.js is simpler, D3.js offers most control but is complex). Ensure interactivity (zoom, pan, hover).
5. **Refine the Plan Structure:**
    - Start with a high-level summary (Goal, User, Concept).
    - Detail each Block: Left Pane (UI/Controls), Right Pane (Visualization), Key Logic/Backend Tasks.
    - Add a dedicated "Technical Architecture" section discussing stack, deployment (local server/Electron), communication (Node-Python), persistence, visualization library choice.
    - Include "Key Challenges" (Python dependency, long computations, state management complexity).
    - Outline "Development Steps" (setup, block-by-block implementation, backend integration, testing, packaging).
    - Emphasize iterative development and user feedback.
    - Explicitly confirm understanding and ask for clarification.
6. **Review and Edit:** Ensure clarity, consistency, and address all points from the original request. Use clear headings and bullet points. Make it sound like a feasible plan, acknowledging complexities. Check the tone – supportive and understanding of the non-technical user's goal.

Ок, Напиши мне список из 30 шагов, выполняя которые я смогу реализовать такой сайт. Я могу использовать при написании кода для каких-то шагов ИИ - пиши мне конкретно, какие запросы делать к gemini по ходу разработки. Я могу выполнять ТОЛЬКО простейшие действия - скопировать код и вставить его в файл, создать файл.

Okay, let's break down the creation of this application into 30 concrete steps, designed for someone comfortable with copy-pasting but not deep coding. We'll use Node.js for the local server, React for the frontend, and keep calling your Python script logic.

Assumptions:

- You have Node.js and npm installed on your computer. (If not, the first step is to install them from nodejs.org).
- You have Python installed with pandas, numpy, statsmodels, openpyxl (for reading new Excel formats). If not, install them (pip install pandas numpy statsmodels openpyxl).
- We'll use create-react-app for simplicity, although faster tools like Vite exist.

Project Setup Folder: Create a main folder for your project, e.g., WageModelApp. We'll work inside this.

Phase 1: Basic Setup & Structure (Steps 1-7)

1. **Initialize Backend:**
   - Open your computer's terminal (Command Prompt, PowerShell, or Terminal on Mac/Linux).
   - Navigate into your WageModelApp folder: cd path/to/WageModelApp
   - Run: npm init -y (This creates a package.json file).
   - Run: npm install express cors (Installs necessary backend libraries: express for the server, cors to allow communication between frontend/backend).
   - Run: npm install --save-dev nodemon (Installs a tool to automatically restart the server on changes).
2. **Create Basic Backend Server File:**
   - Create a file named server.js in the WageModelApp folder.
   - **Copy and Paste the following code into server.js:**

```
      const express = require('express');
const cors = require('cors');
const path = require('path'); // Needed for later file serving

const app = express();
const port = 5001; // Port for the backend server

app.use(cors()); // Allow requests from the frontend
app.use(express.json({ limit: '50mb' })); // Allow reading JSON data in requests, increased limit
app.use(express.urlencoded({ limit: '50mb', extended: true })); // Allow reading form data, increased limit

// Simple test route
app.get('/api/test', (req, res) => {
  res.json({ message: 'Backend is running!' });
});

// --- Routes for different blocks will be added here ---

app.listen(port, () => {
  console.log(`Backend server listening at http://localhost:${port}`);
});
```

3. **Add Start Script for Backend:**
   - Open the package.json file in WageModelApp.
   - Find the "scripts" section.
   - **Modify it to look like this (add the "server" line):**

```
      "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "server": "nodemon server.js"
  },
```

IGNORE_WHEN_COPYING_START

content_copy download

Use code with caution.Json

IGNORE_WHEN_COPYING_END

4. **Initialize Frontend:**
    - In the terminal, *still inside the WageModelApp folder*, run:
      npx create-react-app client
    - This will create a new folder named client with the React frontend code. It might take a few minutes.
5. **Install Frontend Communication Library:**
    - Navigate into the new frontend folder: cd client
    - Run: npm install axios (Installs library for making requests to the backend).
6. **Clean Up Frontend:**
    - Go into the WageModelApp/client/src folder.
    - Delete the files: App.test.js, logo.svg, reportWebVitals.js, setupTests.js.
    - Open App.css and delete all its content.
    - Open index.css and delete all its content.
7. **Basic Frontend Structure:**
    - Open WageModelApp/client/src/App.js.
    - **Replace its entire content with this:**

```
      import React, { useState } from 'react';
import './App.css';
import axios from 'axios'; // Import axios

function App() {
  const [backendMessage, setBackendMessage] =
useState('Loading...');

  // Test connection to backend on component mount
  React.useEffect(() => {
    axios.get('http://localhost:5001/api/test') // Use correct
backend address
      .then(response => {
        setBackendMessage(response.data.message);
      })
      .catch(error => {
        setBackendMessage('Error connecting to backend. Is it
running?');
        console.error("Backend connection error:", error);
      });
  }, []);

  return (
    <div className="App" style={{ display: 'flex', height:
'100vh' }}>
      {/* Left Pane */}
      <div className="left-pane" style={{ width: '40%',
borderRight: '1px solid #ccc', padding: '10px', overflowY: 'auto'
}}>
        <h2>Controls</h2>
        <p>Backend Status: {backendMessage}</p>
        {/* Block 1 Controls will go here */}
        <hr />
        {/* Block 2 Controls will go here */}
        <hr />
        {/* Block 3 Controls will go here */}
        <hr />
        {/* Block 4 Controls will go here */}
      </div>

      {/* Right Pane */}
      <div className="right-pane" style={{ width: '60%', padding:
'10px', overflowY: 'auto' }}>
        <h2>Visualization / Output</h2>
```

```
          {/* Visualizations will go here */}
        </div>
      </div>
    );
}

export default App;
```

IGNORE_WHEN_COPYING_START

content_copy download

Use code with caution.JavaScript

IGNORE_WHEN_COPYING_END

- Open WageModelApp/client/src/index.js.
- **Replace its entire content with this:**

```
      import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

IGNORE_WHEN_COPYING_START

content_copy download

Use code with caution.JavaScript

IGNORE_WHEN_COPYING_END

Phase 2: Block 1 - Data Import (Steps 8-12)

1. **Create Python Script Folder:**
   - In the main WageModelApp folder, create a new folder named python_scripts.
2. **Create Python Script for Initial Load:**
   - Create a file named step1_load_data.py inside the python_scripts folder.
   - **Gemini Prompt:** "Create a Python script using pandas and openpyxl that takes an Excel file path as a command-line argument. It should read the first sheet, identify potential time series columns (assuming the first column 'A' is dates and subsequent columns are data starting from row 7, with metadata in rows 1-6 like in the WagesModel.py example). It should infer frequency if possible. The script should output a JSON string to standard output containing a list of detected series, each with 'name' (from metadata description or column index), 'frequency' (e.g., 'Q', 'M', 'Unknown'), 'metadata' (a dictionary of the first 6 rows), and the actual

'data' as a list of [timestamp, value] pairs. Handle potential errors during file reading or processing and output an error JSON like {'error': 'message'}."
    - **Copy and Paste the generated Python code into step1_load_data.py.** *(Self-correction: Need error handling and specific output format)*
3. **Create Backend Endpoint for File Upload:**
    - Open WageModelApp/server.js.
    - Add const { spawn } = require('child_process'); and const fs = require('fs'); and const multer = require('multer'); at the top with other require statements.
    - Configure multer for temporary file storage:
    - ```
      // Configure temporary storage for uploaded files
      const upload = multer({ dest: 'uploads/' }); // Make sure 'uploads' folder exists
      if (!fs.existsSync('uploads')) {
          fs.mkdirSync('uploads');
      }
      ```

      IGNORE_WHEN_COPYING_START

      content_copy download

      Use code with caution.JavaScript

      IGNORE_WHEN_COPYING_END

    - **Add the following endpoint code inside server.js (before app.listen):**
    - ```
      // Endpoint for Step 1: Upload Excel and get initial series list
      app.post('/api/upload', upload.single('excelFile'), (req, res) => {
        if (!req.file) {
          return res.status(400).json({ error: 'No file uploaded.' });
        }

        const pythonScriptPath = path.join(__dirname, 'python_scripts', 'step1_load_data.py');
        const filePath = req.file.path; // Path to the temporarily uploaded file

        console.log(`Calling Python script: ${pythonScriptPath} with file ${filePath}`);
        const pythonProcess = spawn('python', [pythonScriptPath, filePath]);

        let scriptOutput = '';
        let errorOutput = '';

        pythonProcess.stdout.on('data', (data) => {
          scriptOutput += data.toString();
        });

        pythonProcess.stderr.on('data', (data) => {
          errorOutput += data.toString();
        });

        pythonProcess.on('close', (code) => {
          // Clean up the uploaded file
          fs.unlink(filePath, (err) => {
            if (err) console.error("Error deleting temp file:", err);
          });
      ```

```
    if (code !== 0) {
      console.error(`Python script error (code ${code}): ${errorOutput}`);
      return res.status(500).json({ error: `Python script failed: ${errorOutput}` });
    }

    try {
      const result = JSON.parse(scriptOutput);
      if (result.error) {
         console.error("Python script reported error:", result.error);
         return res.status(400).json(result);
      }
      console.log("Python script success, sending data to frontend.");
      res.json(result); // Send the parsed JSON back to the frontend
    } catch (e) {
      console.error('Error parsing Python output:', e);
      console.error('Raw Python output:', scriptOutput);
      res.status(500).json({ error: 'Failed to parse data from analysis script.' });
    }
  });
});
```

IGNORE_WHEN_COPYING_START

content_copy download

Use code with caution.JavaScript

IGNORE_WHEN_COPYING_END

4. **Create Frontend File Input Component:**
   - Create a new folder WageModelApp/client/src/components.
   - Create a file Block1_DataImport.js inside components.
   - **Gemini Prompt:** "Generate a React functional component named Block1_DataImport. It should contain: 1. An <input type='file' accept='.xlsx, .xls' /> element. 2. State variables using useState to store the list of detected series (detectedSeries), loading status (isLoading), and error messages (error). 3. An onChange handler for the file input that takes the selected file, creates FormData, appends the file, and sends it via POST request to http://localhost:5001/api/upload using axios. It should handle the loading state and update detectedSeries or error based on the response. 4. A section to display 'Loading...' when isLoading is true. 5. A section to display the error message if it exists. 6. A section that maps over detectedSeries (if it's an array) and renders a simple 'plaque' (a div) for each series, displaying its 'name' and 'frequency'. Add basic styling."
   - **Copy and Paste the generated React code into Block1_DataImport.js.**
5. **Integrate Block 1 Component into App:**
   - Open WageModelApp/client/src/App.js.
   - Add import Block1_DataImport from './components/Block1_DataImport'; at the top.

- Inside the left-pane div, replace the comment {/* Block 1 Controls will go here */} with <Block1_DataImport />.

Phase 3: Block 1 - Visualization & Selection (Steps 13-15)

1. **Add Charting Library:**
   - In the terminal, navigate to WageModelApp/client: cd path/to/WageModelApp/client
   - Run: npm install chart.js react-chartjs-2
2. **Create Visualization Component:**
   - Create a file TimeSeriesChart.js in WageModelApp/client/src/components.
   - **Gemini Prompt:** "Create a React functional component TimeSeriesChart that accepts a seriesData prop. This prop should be an object with 'name' and 'data' (an array of [timestamp, value] pairs). Use chart.js and react-chartjs-2 (specifically the Line component) to display a line chart of the time series data. The x-axis should represent time, and the y-axis the value. Make the chart responsive and include basic tooltips. Handle the case where seriesData is null or empty by showing a placeholder message."
   - **Copy and Paste the generated React code into TimeSeriesChart.js.**
3. **Connect Block 1 Plaques to Visualization:**
   - Open WageModelApp/client/src/App.js.
   - Add import TimeSeriesChart from './components/TimeSeriesChart';
   - Add state for the currently viewed series: const [viewingSeries, setViewingSeries] = useState(null);
   - Pass setViewingSeries down as a prop to Block1_DataImport: <Block1_DataImport setViewingSeries={setViewingSeries} />.
   - In the right-pane div, replace the comment {/* Visualizations will go here */} with <TimeSeriesChart seriesData={viewingSeries} />.
   - Modify WageModelApp/client/src/components/Block1_DataImport.js:
     - Accept the setViewingSeries prop: function Block1_DataImport({ setViewingSeries }) { ... }
     - Inside the .map function where plaques are rendered, add an onClick handler to each plaque's div: onClick={() => setViewingSeries(series)}. Make sure series in the map callback holds the full series object including the 'data' field. *You might need to adjust the Python script (Step 9) and backend endpoint (Step 10) to ensure the full series data is sent to the frontend initially.*

Phase 4: Block 1 - Aggregation & Passing Data (Steps 16-18)

1. **Add Selection and Aggregation UI to Block 1:**
   - Modify WageModelApp/client/src/components/Block1_DataImport.js:
     - Add state to track selected series IDs (e.g., using the series name or an index): const [selectedSeriesIds, setSelectedSeriesIds] = useState(new Set());
     - Add a checkbox to each series plaque. When checked/unchecked, update the selectedSeriesIds state.
     - Add a dropdown menu for selecting the target timeframe (e.g., 'Q', 'M'). Store the selection in state: const [targetTimeframe, setTargetTimeframe] = useState('Q');
     - Add a "Process Selected Series" button.
2. **Create Python Script for Aggregation:**

- Create step1b_aggregate_data.py in python_scripts.
- **Gemini Prompt:** "Create a Python script that accepts JSON data via standard input. The JSON should contain: target_timeframe (e.g., 'Q') and series_list (an array of series objects, each with 'name', 'data' as [timestamp, value] pairs, and 'frequency'). The script should iterate through series_list, convert the 'data' into a pandas Series with a DatetimeIndex, resample each series to the target_timeframe (using mean for aggregation from finer to coarser, e.g., M to Q). Handle potential errors. Output the processed data as JSON to standard output, in the format {'processed_data': {'series_name1': [[ts, val], ...], 'series_name2': [[ts, val], ...]}} or {'error': 'message'}."
- **Copy and Paste the generated Python code into step1b_aggregate_data.py.**

3. **Create Backend Endpoint and Frontend Logic for Aggregation:**
   - In WageModelApp/server.js, add a new endpoint /api/process_series.
     - **Gemini Prompt:** "Generate Node.js code for an Express endpoint /api/process_series (POST). It should: 1. Receive JSON data in the request body containing target_timeframe and series_list. 2. Call the Python script python_scripts/step1b_aggregate_data.py using child_process.spawn. 3. Pass the received JSON data to the Python script via standard input (pythonProcess.stdin.write(...), pythonProcess.stdin.end()). 4. Read the resulting JSON from the Python script's standard output. 5. Send the processed data or an error back to the frontend."
     - **Copy and Paste the generated Node.js endpoint code into server.js.**
   - In WageModelApp/client/src/components/Block1_DataImport.js, add an onClick handler to the "Process Selected Series" button.
     - This handler should filter detectedSeries based on selectedSeriesIds, create the JSON payload ({ target_timeframe, series_list }), and POST it to /api/process_series.
     - On success, it should probably pass the processed_data up to the App component (using another prop function passed down from App).
   - In WageModelApp/client/src/App.js, add state to hold the processed data from Block 1: const [processedData, setProcessedData] = useState(null);. Pass setProcessedData down to Block1_DataImport.

Phase 5: Block 2 - Data Enrichment (Steps 19-21)

1. **Create Block 2 UI Component:**
   - Create Block2_DataEnrichment.js in WageModelApp/client/src/components.
   - **Gemini Prompt:** "Generate a React functional component Block2_DataEnrichment that receives processedData (from Block 1, format: {'series_name1': [[ts, val], ...], ...}) and a function updateEnrichedData as props. It should: 1. Display a list of series names from processedData. 2. For each series, provide buttons/options for 'Rename', 'Differentiate (Abs)', 'Differentiate (%)', 'Normalize by...'. 3. Maintain its own state enrichedData, initialized from processedData. 4. When an action is performed (e.g., Differentiate), call a backend endpoint (to be created) to perform the calculation and update enrichedData. 5. Call updateEnrichedData whenever enrichedData changes. 6. Also display the series in the right-pane chart when clicked (pass setViewingSeries down from App)."
   - **Copy and Paste the generated React code into Block2_DataEnrichment.js.**
   - In App.js, add state const [enrichedData, setEnrichedData] = useState(null);, pass processedData and setEnrichedData to Block2_DataEnrichment, and render it in

the left pane below Block 1 (conditionally, maybe only after processedData exists).

2. **Create Python Script(s) for Enrichment:**
    - Create step2_enrich_series.py in python_scripts.
    - **Gemini Prompt:** "Create a Python script step2_enrich_series.py that accepts JSON via stdin. The JSON should specify an operation ('diff_abs', 'diff_pct', 'normalize'), the series_data ([[ts, val], ...]), and potentially period (integer) or denominator_data ([[ts, val], ...]). The script should perform the requested pandas operation (.diff(periods=period), .pct_change(periods=period) * 100, or division) on the input series data. Output the resulting series data as JSON {'result_data': [[ts, val], ...]} or {'error': 'message'}."
    - **Copy and Paste the generated Python code.**
3. **Create Backend Endpoint(s) for Enrichment:**
    - In WageModelApp/server.js, add an endpoint /api/enrich.
    - **Gemini Prompt:** "Generate Node.js code for an Express endpoint /api/enrich (POST). It should receive JSON specifying the operation and necessary data (series, period, denominator). It should call python_scripts/step2_enrich_series.py, pass the data via stdin, get the result, and send it back to the frontend."
    - **Copy and Paste the generated Node.js code.** Connect the buttons in Block2_DataEnrichment.js to call this endpoint.

Phase 6: Block 3 - Model Configuration & Run (Steps 22-26)

1. **Create Block 3 UI Component:**
    - Create Block3_Modeling.js in WageModelApp/client/src/components.
    - **Gemini Prompt:** "Generate a React functional component Block3_Modeling that receives enrichedData (from Block 2) as a prop. It should allow the user to: 1. Select a 'Dependent Variable' from a dropdown of series names in enrichedData. 2. Display the remaining series names as potential 'Regressors' with options for each (Include/Exclude/Required, Sign constraint). 3. Select Constant option (Include/Exclude/Test). 4. Input Max Lag depth. 5. Select Statistical Tests (checkboxes for VIF, p-value threshold). 6. Select Accuracy Metrics (checkboxes for R^2, Adj R^2, MAPE). 7. Display the estimated number of models to test. 8. Have a 'Run Model Search' button."
    - **Copy and Paste the generated React code.** Render it in App.js below Block 2.
2. **Create Python Script for Model Fitting:**
    - Create step3_run_regression.py in python_scripts.
    - **Gemini Prompt:** "Create a Python script step3_run_regression.py that accepts JSON via stdin. The JSON should contain: dependent_var_name, dependent_var_data, regressors (a dictionary where keys are regressor names and values are their data [[ts, val], ...]), lags (a dictionary mapping regressor names to specific lag values for this run), include_constant (boolean), vif_threshold, p_value_threshold. The script should: 1. Prepare the dependent variable and lagged regressor data using pandas, aligning indices and dropping NaNs. 2. Add a constant if include_constant is true. 3. Fit an OLS model using statsmodels.api.OLS. 4. Calculate R^2, Adj R^2, MAPE. 5. Check if all regressor p-values are below p_value_threshold. 6. Calculate VIF for all regressors (if more than one) and check if all are below vif_threshold. 7. Output results as JSON including coefficients, p_values, metrics (r_squared, adj_r_squared, mape), test_results (p_value_ok, vif_ok), n_obs, and valid (boolean, true if all selected tests pass)."

    - **Copy and Paste the generated Python code.**
3. **Create Backend Logic for Model Search Orchestration:**
    - In WageModelApp/server.js, add state (outside endpoints, maybe just a simple in-memory object for now) to store model results: let modelResultsStore = {}; let currentJobId = null;
    - Add an endpoint /api/start_search (POST).
        - It receives the configuration from Block 3 UI (dependent var, regressors config, lags, tests, metrics).
        - It generates ALL combinations of regressors (respecting Include/Exclude/Required flags later) and lags (0 to N for each included regressor). *This generation logic is complex.*
        - **Gemini Prompt (for the combination logic):** "Write a Javascript function that takes a list of potential regressor names, a max lag N, and an optional list of required regressors. It should generate all possible model specifications, where each specification is an object containing regressors (a list of names included in this model) and lags (a dictionary mapping each included regressor name to a specific lag value between 0 and N). It should consider all combinations of regressors (subsets) and all combinations of their individual lags. Ensure required regressors are always included." *(This will need careful integration)*.
        - It should store these specifications.
        - It starts processing them one by one (or in small batches), calling step3_run_regression.py for each specification. Use async/await or promises to manage the sequential calls.
        - As results come back from Python, store them in modelResultsStore keyed by a unique identifier for the specification.
        - It should immediately return a jobId to the frontend so it can poll for progress.
    - Add an endpoint /api/search_progress/:jobId (GET).
        - Returns the current state of modelResultsStore for that jobId (how many done, how many valid, etc.).
    - Add endpoints for /api/pause_search/:jobId and /api/stop_search/:jobId (implementing pause/stop logic in the backend loop).
4. **Implement Frontend Polling and Progress Visualization:**
    - In WageModelApp/client/src/components/Block3_Modeling.js, when "Run" is clicked:
        - Call /api/start_search.
        - Store the received jobId.
        - Start polling /api/search_progress/:jobId periodically (e.g., every 2 seconds) using setInterval and useEffect.
        - Update state based on the progress response (e.g., number processed, number valid).
    - In the right pane (App.js), create a component ModelSearchVisualizer.js.
        - **Gemini Prompt:** "Generate a React component ModelSearchVisualizer that receives model results progress data (e.g., an object mapping specification IDs to their status: 'pending', 'valid', 'invalid_stats', 'invalid_constraints'). Render this as a grid of small squares, colored according to the status (grey, green, red, yellow). Make it update dynamically as the progress data changes."
        - Render this component in the right pane when a search is active.
5. **Implement Basic Persistence (Backend):**

- Modify the /api/start_search endpoint in server.js. Before starting, check if results for this configuration (maybe hash the config?) already exist in a JSON file (e.g., results_cache/config_hash.json). If so, load them into modelResultsStore.
- Modify the result handling part: After receiving a result from Python, append it to both the in-memory modelResultsStore and the corresponding JSON file using fs.appendFile (or read-modify-write). This ensures progress is saved.
- *Note: Implementing full "Realities" is more complex and might require a more structured way to save/load different configurations and their associated result files.*

Phase 7: Block 4 - Results Viewer (Steps 27-29)

1. **Create Block 4 UI Component:**
   - Create Block4_ModelViewer.js in WageModelApp/client/src/components.
   - **Gemini Prompt:** "Generate a React component Block4_ModelViewer that receives the final modelResults (the full store from the backend after the search finishes). It should: 1. Filter the results to show only 'valid' models. 2. Allow selecting a metric (R^2, Adj R^2, MAPE) to sort/rank models. 3. Display the model space visualization again, but this time as a heatmap based on the selected metric for valid models only. 4. Allow clicking on a valid model square to select it. 5. Display detailed stats (coefficients, p-values, metrics) for the selected model below the heatmap."
   - **Copy and Paste the generated React code.** Render in App.js below Block 3.
2. **Create Decomposition Chart Component:**
   - Create DecompositionChart.js in WageModelApp/client/src/components.
   - **Gemini Prompt:** "Generate a React component DecompositionChart using chart.js and react-chartjs-2. It should accept actualData ([[ts, val], ...]), predictedData ([[ts, val], ...]), and contributions (an object where keys are regressor names and values are their contribution series [[ts, val], ...]). Render a stacked bar chart showing the contributions, overlaid with line charts for actualData and predictedData. Ensure correct time alignment on the x-axis."
   - **Copy and Paste the generated React code.** Render this in the right pane in App.js, driven by the model selected in Block4_ModelViewer.
3. **Connect Block 4 UI to Decomposition Chart:**
   - In App.js, add state for the model selected in Block 4: const [selectedFinalModel, setSelectedFinalModel] = useState(null);. Pass the setter down to Block4_ModelViewer.
   - When a model is selected in Block4_ModelViewer, it needs to fetch or calculate the necessary data (actual Y, predicted Y, contributions) for that specific model. This might require another backend endpoint /api/get_model_details/:specId that potentially re-runs the specific Python script or retrieves saved detailed predictions.
   - Pass the data needed by DecompositionChart to it based on selectedFinalModel.

Phase 8: Running the App (Step 30)

1. **Run Locally:**
   - You need two terminals open.
   - **Terminal 1 (Backend):**
     - Navigate to WageModelApp: cd path/to/WageModelApp
     - Run: npm run server (This starts the Node.js backend on http://localhost:5001)

- **Terminal 2 (Frontend):**
  - Navigate to WageModelApp/client: cd path/to/WageModelApp/client
  - Run: npm start (This starts the React development server, usually opening http://localhost:3000 in your browser).
- Use the application in your browser!

This is a detailed roadmap. Each step involves creating files and copy-pasting code, often generated by Gemini based on the specific prompts. Remember to adjust file paths and variable names if needed. Debugging will likely be necessary, checking terminal outputs from both the backend (npm run server) and the frontend (npm start, and the browser's developer console). Good luck!