

# Kai Health: Annotated AI agent interaction

Gordon Lemmon

## Table of Contents

<b>Table of Contents.....</b>	<b>1</b>
<b>Background.....</b>	<b>2</b>
<b>The Problem.....</b>	<b>3</b>
<b>The steering documents.....</b>	<b>3</b>
product.md.....	3
tech.md.....	3
structure.md.....	4
<b>Development of project specifications.....</b>	<b>4</b>
ChatGPT prompt 1 (requirements.md).....	4
ChatGPT Response 1 (requirements.md).....	5
ChatGPT prompt 2 (Framingham.csv).....	5
ChatGPT response 2 (requirements.md - preloaded data).....	5
ChatGPT prompt 3 (test.md).....	5
ChatGPT response 3 (test.md).....	5
ChatGPT prompt 4 (Framingham.csv).....	6
ChatGPT response 4 (requirements.md - baked in data).....	7
ChatGPT prompt 5 (test.md - baked in data).....	8
ChatGPT response 5 (test.md - baked in data).....	8
ChatGPT prompt 6 (write the test code).....	8
ChatGPT response 6 (test code).....	8
ChatGPT Prompt 7 (design.md).....	8
ChatGPT Response 7 (design.md).....	9
ChatGPT Prompt 8 (requirements.md with updated python libraries).....	9
ChatGPT Response 8 (requirements.md - library updates).....	9
ChatGPT Prompt 9 (design.md with updated python libraries).....	15
ChatGPT Prompt 9 (design.md with updated python libraries).....	15
ChatGPT Prompt 10: tasks.md.....	23
ChatGPT Response 10: tasks.md.....	23
ChatGPT Prompt 11 (write the code).....	32
ChatGPT Response 11 (written code).....	32
ChatGPT Prompt 12 (flesh out the tests).....	32
ChatGPT Response 12 (written code with additional tests).....	32

ChatGPT Prompt 13 (add index.html).....	33
ChatGPT Response 13 (index.html provided).....	33
ChatGPT Prompt 14.....	34
ChatGPT Response 14.....	34
ChatGPT Prompt 15 (host the website).....	35
ChatGPT Response 15 (website hosting instructions).....	35
ChatGPT Prompt 16.....	37
ChatGPT Response 16.....	37
<b>Conclusion.....</b>	<b>37</b>

## Background

When it comes to optimizing interactions with an AI agent, context is paramount. I am inspired by Kiro's spec driven development. Kiro is an agentic IDE developed by a team at Amazon Web Services. Kiro moves developers beyond “vibe” coding, toward structured documents that clearly define project requirements and breaking down the problem into subtasks. Kiro’s steering documents “remind” the forgetful AI agent of your coding conventions and established patterns and libraries. They help establish “persistent” knowledge when none such exists with current AI agents. As AI agent context window size continues to increase, this reminding may become obsolete, but the documents themselves will continue to help developers contextualize the project for the AI agent.

Kiro suggests 3 steering documents:

- [product.md](#) defines purpose, target users, business objectives
- [tech.md](#) documents frameworks, libraries, development tools, and technical constraints
- [structure.md](#) outlines file organization, naming conventions, import patterns, and architectural decisions.

With steering docs in place, the developer collaborates with the AI agent to write 3 project specification files:

- [requirements.md](#) records user stories and acceptance criteria in structured form
- [design.md](#) captures architecture, diagrams, and implementation considerations
- [tasks.md](#) breaks the design into discrete, trackable coding steps

While Kiro is not yet available, the concepts are easily implemented in any chat agent. By providing the context above at the beginning of each chat session, the agent “remembers”. Of course these documents need not be static. A dynamic feedback loop can be used to improve these documents as we build the application. In fact, rather than envisioning our communication with the AI agent as restricted to the software design process, we can broaden our view to include the entire software development lifecycle (SDLC). Considering for instance, step 5 of SDLC (testing), “[tech.md](#)” could specify a unit test framework to use, and “[tasks.md](#)” could

require the writing of tests. As we will explore, the AI agent can also help us write these documents.

## The Problem

For the sake of simplicity I decided to demonstrate these principles in a toy example, using data from the Framingham heart study, which tracked patient outcomes from 1948 to 1968. The goal is to build a browser-based tool to explore this data.

## The steering documents

### [product.md](#)

#### Purpose

Provide a web-interface for browsing cardiology survival curves and proportional hazards as found in the CSV file Framingham.csv

#### Target users

Research scientists

#### Objectives

1. For each outcome, show the effect of each predictor variable on hazards rate, using Cox-proportional hazards
2. For each outcome, show the unadjusted Kaplan-Meier survival curve

### [tech.md](#)

#### Frameworks

- This code should be written in python
- The Flask Framework should be used to render the http front-end
- The PyTest unit testing framework should be used for unit testing

## Libraries

- The python libraries Pandas, Numpy, and Scipy should be used for data science.
- You may use Lifelines and Scikit-survival (sksurv) libraries for survival analysis
- Matplotlib may be used for plots and visualizations

## Technical constraints

- For this project, use HTTP not HTTPS
- Build this website as a single page
- Assume the server has access to a single CPU, no GPUs
- Write unit tests for each function

## [structure.md](#)

### Conventions

- Follow all guidelines published as PEP 8 style guide
- Provide type hinting (PEP 484) for all variables and functions
- Group imports by (1) standard library, (2) third party, and (3) local modules
- Within each import group, sort alphabetically
- Add clear and concise docstrings to modules, classes, and functions following PEP 257

## Development of project specifications

For the prompts below I'm using Kiro's file descriptions.

I'm using ChatGPT coding assistant 5 (<https://chatgpt.com/g/g-vK4oPfjfp-coding-assistant>)

The idea behind these initial prompts is to help the AI agent document its plan before writing any code

### ChatGPT prompt 1 ([requirements.md](#))

Write a file called [requirements.md](#). This file is a product definition that outlines what will be built in clear language for all stakeholders.

PROMPT: Build a browser based tool for exploring Kaplan-Meier survival curves and Cox-Proportional hazards from the Framingham.csv file.

Key components of requirements.md

- User Stories: These describe the desired functionality from the perspective of an end-user. Generate these stories from my PROMPT
- Acceptance Criteria: Written in EARS (Easy Approach to Requirements Syntax) notation, these outline the specific conditions that must be met for a user story to be considered complete. This includes coverage for edge cases and other conditions.
- Success Metrics: These define what success looks like for the feature being built.

## ChatGPT Response 1 ([requirements.md](#))

[Omitted for Brevity - final version is shown further down]

## ChatGPT prompt 2 (Framingham.csv)

The web-server will provide the Framingham.csv file, so the requirements you came up with related to user input are not relevant.

Recreate the entire requirements.md file but this time, understand that the web-server will provide the dataset ("Framingham.csv") - the user will not upload data.

## ChatGPT response 2 ([requirements.md](#) - preloaded data)

[Omitted for Brevity - final version is shown further down]

## ChatGPT prompt 3 ([test.md](#))

Since I believe in test driven development, let's generate a matching test plan (PyTest outline + fixtures) and a UI wireframe sketch.

## ChatGPT response 3 ([test.md](#))

[Omitted for brevity - final version is shown further down]

## ChatGPT prompt 4 (Framingham.csv)

Before we go any further, let me provide you with the file Framingham.csv and explain its columns.

[File attached]

### **Participant demographics and exam data**

- RANDID: A unique identification number assigned to each participant.
- SEX: The participant's sex, coded as 1 for Male and 2 for Female.
- AGE: The participant's age in years at the time of the exam.
- TOTCHOL: Serum total cholesterol level in mg/dL.
- SYSBP: Systolic blood pressure in mmHg, averaged from the last two of three measurements.
- DIABP: Diastolic blood pressure in mmHg, averaged from the last two of three measurements.
- CURSMOKE: A binary indicator (0 or 1) of whether the participant is a current smoker.
- CIGPDAY: The number of cigarettes smoked per day. This is 0 for non-smokers.
- BMI: Body Mass Index, calculated as weight in kilograms divided by height in meters squared.
- DIABETES: A binary indicator (0 or 1) of whether the participant is diabetic.
- BPMEDS: A binary indicator (0 or 1) of whether the participant was using anti-hypertensive medication at the time of the exam.
- HEARTRTE: Heart rate in beats per minute.
- GLUCOSE: Casual serum glucose level in mg/dL.
- educ: The participant's education level. This variable is typically coded into categories such as some high school, high school/GED, some college/vocational school, or college degree.
- PERIOD: The examination cycle (1, 2, or 3) during which the data was collected.
- TIME: The number of days since the baseline exam.
- HDLC: High-density lipoprotein cholesterol (HDL-C) in mg/dL, available for the third exam period only.
- LDLC: Low-density lipoprotein cholesterol (LDL-C) in mg/dL, also available for the third exam period only.

### **Prevalent (pre-existing) conditions at the time of the exam**

- PREVCHD: A binary indicator (0 or 1) for prevalent coronary heart disease.
- PREVAP: A binary indicator (0 or 1) for prevalent angina pectoris.
- PREVMI: A binary indicator (0 or 1) for prevalent myocardial infarction.
- PREVSTRK: A binary indicator (0 or 1) for prevalent stroke.
- PREVHYP: A binary indicator (0 or 1) for prevalent hypertension.
- Follow-up outcomes (tracked over 24 years)
- DEATH: A binary indicator (0 or 1) for death from any cause during the follow-up period.

- ANGINA: A binary indicator (0 or 1) for the occurrence of angina pectoris during the follow-up.
- HOSPMI: A binary indicator (0 or 1) for the occurrence of hospitalized myocardial infarction during the follow-up.
- MI\_FCHD: A binary indicator (0 or 1) for hospitalized myocardial infarction or fatal coronary heart disease during the follow-up.
- ANYCHD: A binary indicator (0 or 1) for any coronary heart disease event during the follow-up.
- STROKE: A binary indicator (0 or 1) for the occurrence of a stroke event during the follow-up.
- CVD: A binary indicator (0 or 1) for any cardiovascular disease event during the follow-up.
- HYPERTEN: A binary indicator (0 or 1) for the development of hypertension during the follow-up.

### Time-to-event outcomes

- TIMEAP: The number of days from the baseline exam to the first angina event or the censoring date.
- TIMEMI: The number of days from the baseline exam to the first hospitalized MI event or the censoring date.
- TIMEMIFC: The number of days from the baseline exam to the first MI or fatal CHD event or the censoring date.
- TIMECHD: The number of days from the baseline exam to the first ANYCHD event or the censoring date.
- TIMESTRK: The number of days from the baseline exam to the first stroke event or the censoring date.
- TIMECVD: The number of days from the baseline exam to the first CVD event or the censoring date.
- TIMEDTH: The number of days from the baseline exam to death or the censoring date.
- TIMEHYP: The number of days from the baseline exam to the first hypertension event or the censoring date.

## ChatGPT response 4 ([requirements.md](#) - baked in data)

Awesome—thanks for the dataset and the detailed column descriptions. I've rebuilt the **requirements.md** end-to-end so it assumes the **server provides Framingham.csv** (no uploads) and it now bakes in your actual column names, defaults, and edge cases.

[Omitted for brevity - final version is shown further down]

If you want, I can now regenerate the **test plan** to align 1:1 with this doc (including new checks

for prevalent-disease exclusions, period-3 lipid handling, and smoking consistency), or jump straight to a **tests/ scaffold** with PyTest stubs and fixture templates.

## ChatGPT prompt 5 ([test.md](#) - baked in data)

Please regenerate the test plan

## ChatGPT response 5 ([test.md](#) - baked in data)

Awesome — here's the **regenerated test plan** aligned 1:1 with the updated requirements and the fact that **Framingham.csv** is **preloaded on the server** (no uploads). It adds explicit coverage for **prevalent-disease exclusions**, **period-3 lipid handling (HDLc, LDLc)**, and **smoking consistency (CURSMOKE vs CIGPDAY)**.

[Omitted for brevity - final version is shown further down]

If you want, I can now turn this into a ready-to-run **tests/ scaffold** (with fixture generators, reference calculators, endpoint clients, and test stubs matching the IDs above).

## ChatGPT prompt 6 (write the test code)

Yes, please do.

## ChatGPT response 6 (test code)

Sweet — I built you a ready-to-run **tests/ scaffold** with PyTest, plus a set of small CSV fixtures derived from your actual **Framingham.csv**. It's all written to **/mnt/data/tests**. You can point your Flask app at these during development, or at the full dataset in production.

[Omitted for brevity - final version is shown further down]

## ChatGPT Prompt 7 ([design.md](#))

Let's work on generating a "design.md" document. This document serves as a detailed technical blueprint that outlines how the project will be implemented, and is based on the requirements you've already come up with.



Here are the key contents of design.md:

- System architecture: Outlines the overall structure of the application and how its components interact.
- Data models and schemas: Details the structure of the data, including database schemas and data flow.
- API contracts and interfaces: Specifies the endpoints, request/response formats, and interfaces that components use to communicate.
- Diagrams: Includes architectural or sequence diagrams to visually represent the system's design.
- Implementation considerations: Documents key technical decisions, such as which frameworks or libraries to use and the reasoning behind them.
- Technical assumptions: Explicitly states any assumptions made during the design process.

## ChatGPT Response 7 ([design.md](#))

[Omitted for brevity - final version is shown further down]

## ChatGPT Prompt 8 ([requirements.md](#) with updated python libraries)

I decided to update my guidance on Python libraries. Here is my new guidance:

### Libraries

- The python libraries Pandas, Numpy, Scipy and statsmodels should be used for data science.
- You may use Lifelines and Scikit-survival (sksurv) libraries for survival analysis
- Matplotlib and Seaborn libraries may be used for plots and visualizations
- You may use any other free python libraries, as long as they are published in The Python Package Index (<https://pypi.org/>)

With these libraries available to you, please recreate [requirements.md](#)

## ChatGPT Response 8 ([requirements.md](#) - library updates)

Here's the updated **requirements.md** that incorporates your new Python library guidance while keeping everything aligned with the original product vision, coding conventions, and platform constraints.

## Product Overview

- **Purpose:** Provide a browser-based interface for exploring cardiology survival curves from the Framingham dataset.
- **Target Users:** Research scientists.
- **Core Objectives:**
  1. For each outcome, show the effect of each predictor on the hazard rate using **Cox proportional hazards**. 2) For each outcome, show the **unadjusted Kaplan–Meier** survival curve.
- **Framework & Constraints:** Python + **Flask**; single-page site; **HTTP** (not HTTPS); **single CPU**; **PyTest** for tests.

## Updated Library Guidance (supersedes prior library list)

- Use **Pandas**, **NumPy**, **SciPy**, and **statsmodels** for data work and statistics.
- Use **lifelines** and/or **scikit-survival (sksurv)** for survival analysis (KM, Cox PH).
- Use **Matplotlib** and **Seaborn** for plots and visualizations.
- You may use other **PyPI** packages as needed (e.g., cachetools, pydantic), provided they are free and available on PyPI.  
*(This section reflects your latest guidance and overrides the “Libraries” list in tech.md.)*

## Code Conventions

- Follow **PEP 8**, include **PEP 484** type hints, add **PEP 257** docstrings, and keep imports grouped and alphabetized.

---

## User Stories

1. As a research scientist, I want the tool to **preload Framingham.csv** so I can analyze immediately without uploads.
2. As a research scientist, I want to **choose an outcome** and see an **unadjusted KM curve** (with 95% CI) to understand baseline survival.

3. As a research scientist, I want to **select predictors** and run a **Cox PH model** to get HRs, CIs, and p-values.
  4. As a research scientist, I want **default event/time mappings** per outcome with the ability to **override**.
  5. As a research scientist, I want **clear plots and tables** (labels, legends, captions) for interpretation and export (PNG/SVG).
  6. As a research scientist, I want **filters** (e.g., exclude prevalent disease; restrict to **PERIOD=3** when using **HDLC/LDLC**) and **data quality flags** (e.g., smoking inconsistencies).
  7. As a research scientist, I want **graceful, actionable errors** for convergence, collinearity, or invalid selections.
  8. As a research scientist, I want a **single-page** browser UI that remains responsive during computation.
- 

## Acceptance Criteria (EARS)

### Data & Initialization

- **When** the application starts **then** the server shall load **Framingham.csv** and expose column metadata to the client.
- **When** the dataset is missing or unreadable **then** the UI shall show a blocking error and disable analysis.

### Outcome Mapping & Selection

- **When** the user picks a predefined outcome **then** the system shall auto-map the corresponding event/time columns (with option to override), and this mapping shall appear in the results summary.

### Kaplan–Meier (KM)

- **When** valid time/event columns are set **then** the system shall compute KM using **lifelines** or **sksurv**, returning survival and 95% CI; plot must include axis labels, legend, and title.
- **When** zero events exist **then** the curve shall be flat at 1.0 with an explanatory notice.

- **When** all events occur at time 0 **then** the curve shall be flat at 0.0 with a notice.
- **When** the user hovers the KM plot **then** the tooltip shall display time, survival probability, and CI.

## Cox Proportional Hazards

- **When** time/event and  $\geq 1$  predictors are selected **then** the system shall fit **Cox PH** using **lifelines** or **sksurv**, and display HR, 95% CI, and p-value for each (encoded) predictor.
- **When** categorical predictors are included **then** the system shall one-hot encode them (drop one level as baseline) and report the baseline used.
- **When** collinearity or quasi-separation is detected **then** the system shall list implicated predictors and halt fitting with guidance.
- **When** the model fails to converge **then** the system shall show a non-blocking error with suggestions (e.g., reduce predictors, rescale, merge sparse levels, restrict to **PERIOD=3** for lipids).
- **When** PH diagnostics are requested (basic v1 check) **then** the system shall provide a simple assumption screen or residual summary; full Schoenfeld plots may be out of scope for v1 and must be stated.

## Filters, Missingness, and Data Quality

- **When** modeling incident outcomes (e.g., CHD, stroke) **then** the user shall be able to **exclude prevalent disease** rows (e.g., **PREVCHD=1** for CHD).
- **When** **HDLC/LDLC** are selected without **PERIOD=3** filtering **then** the system shall warn and offer to restrict to **PERIOD=3**.
- **When** inconsistencies (e.g., **CURSMOKE=0** with **CIGPDAY>0**) are detected **then** the system shall report counts and offer to leave/coerce/drop, recording the choice.
- **When** predictors contain missing values **then** the user shall choose to **drop** rows or **impute** (mean/mode), and the method shall be noted in outputs.

## Plot Export & Reporting

- **When** the user clicks **Download Plot** **then** the current KM or Cox plot shall download as **PNG** or **SVG** (Matplotlib/Seaborn).

- **When** the user opens **Results Summary** **then** the system shall list dataset filters, mapping, encoding decisions, tie method, and missingness handling.

## UI & Responsiveness

- **When** the page loads **then** the entire experience shall be a **single-page** interface served by Flask over **HTTP**.
  - **When** long computations run **then** the UI shall show a busy indicator, remain responsive, and retain prior results.
- 

## Success Metrics

- **Usability:**  $\geq 90\%$  of pilot users can (select outcome  $\rightarrow$  view KM  $\rightarrow$  add predictors  $\rightarrow$  run Cox  $\rightarrow$  export) without assistance.
  - **Accuracy:** KM survival values and Cox HR/CI/p match **lifelines/sksurv** results on reference fixtures within agreed tolerance ( $< 1\%$  or numerical tolerance).
  - **Performance (single CPU):** KM  $\leq 2s$ ; Cox ( $\leq \sim 15$  predictors incl. dummies)  $\leq 5s$  on the full dataset.
  - **Reliability:** 100% of specified error paths (missing dataset, non-convergence, collinearity, period-3 lipid warnings, smoking inconsistencies) produce clear messages.
  - **Adoption:**  $\geq 3$  internal research teams use the tool within the first quarter after release.
- 

## Non-Functional Requirements (NFRs)

### Architecture & Environment

- **Flask** server over **HTTP** (no HTTPS in scope); **single-page** UI; **single CPU** only.
- Dataset is bundled with the server; **no user uploads**.

### Libraries & Tooling

- **Core:** Pandas, NumPy, SciPy, **statsmodels**.
- **Survival:** **lifelines** (primary) and/or **sksurv**.

- **Plots:** Matplotlib (+ Seaborn aesthetics).
- **Tests:** PyTest.
- **Optional PyPI utilities:** cachetools (LRU), pydantic (request validation), etc.

## Code Quality & Maintainability

- PEP 8; **PEP 484** type hints for public functions/variables; **PEP 257** docstrings; grouped & alphabetized imports.
- Unit tests  $\geq 85\%$  coverage for validation, KM, Cox, encoding, filters, and error paths.

## Performance & Caching

- Preload dataset; memoize recent KM/Cox results (e.g., LRU cache keyed by selections/filters).
- Avoid  $O(n^2)$  operations; vectorize where feasible.

## Reliability, Logging, Observability

- Structured logs (request id, action, duration, N, events, filters, errors).
- Health endpoint (`/health`) for ops checks.
- Actionable error messages; stack traces only in logs.

## Security & Privacy

- Internal deployment; HTTP only (documented constraint).
- No PII processing beyond the dataset; no persistent user data.

## Accessibility & UX

- Keyboard navigation, ARIA roles on tabs/plots, labeled controls.
- Clear legends, titles, and units; empty-state guidance; inline hints for mappings and filters.

## Out of Scope (v1)

- HTTPS and authentication.
  - User uploads or multiple datasets.
  - Full PH diagnostics (e.g., complete Schoenfeld residual workflows), time-varying covariates, competing risks.
- 

## Assumptions

- `Framingham.csv` ships with the server and loads at startup.
- Events are binary {0,1}; times are days  $\geq 0$ .
- Scientists understand core survival concepts; the UI provides brief context help.
- **Library availability:** lifelines, sksurv, statsmodels, Matplotlib, and Seaborn (and any additional PyPI packages you approve) will be available in the runtime environment.

## ChatGPT Prompt 9 ([design.md](#) with updated python libraries)

Update [design.md](#) now that I have expanded our python library guidance.

## ChatGPT Prompt 9 (design.md with updated python libraries)

I've regenerated both the [design.md](#) with your updated library guidance (lifelines / statsmodels, with room for sksurv). It preloads `Framingham.csv`, serves a single-page UI, computes KM with `lifelines.KaplanMeierFitter`, and runs Cox PH with `lifelines.CoxPHFitter` (one-hot encoding, baseline capture, basic warnings for HDLC/LDLC without `PERIOD=3`, etc.). Exports plots via Matplotlib.

## [Design.md](#) v1.1

### 0) What changed (vs v1.0)

- **KM & Cox PH:** Implemented with **lifelines** (primary).

- **Optional backend:** **scikit-survival (sksurv)** for verification and future models.
  - **Statsmodels:** supplemental inference/stats if needed.
  - **Plots:** **Matplotlib** backbone; **Seaborn** for improved aesthetics.
  - **PyPI utilities:** allowed (e.g., cachetools, pydantic).
- 

## 1) System Architecture

### 1.1 Overview

- **Client (SPA):** HTML + vanilla JS, fetch API calls. Renders plots/tables, supports downloads.
- **Server (Flask):**
  - **Dataset registry:** load `Framingham.csv`, provide metadata.
  - **Validation/filters:** enforce mappings, handle missingness, apply prevalent-disease and period filters, detect smoking inconsistencies.
  - **KM Engine:** `lifelines.KaplanMeierFitter`.
  - **Cox Engine:** `lifelines.CoxPHFitter` (Efron ties).
  - **Diagnostics:** basic PH checks; detailed diagnostics out of scope v1.
  - **Plotting:** Matplotlib + Seaborn.
  - **Caching:** in-memory (e.g., cachetools LRU).

### 1.2 Module layout

```
app/  
  __init__.py  
  routes.py  
  settings.py
```

```
core/  
  dataset.py  
  mapping.py  
  validate.py  
  filters.py
```

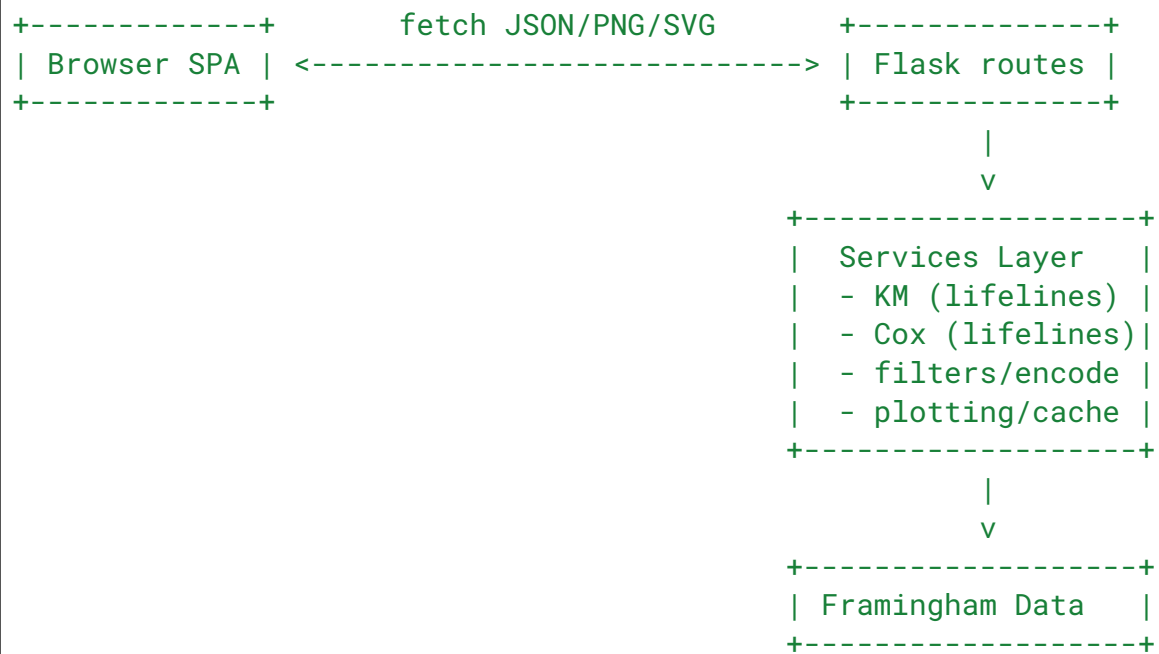


```
km.py
cox.py
encode.py
diagnostics.py
plotting.py
cache.py
```

```
ui/
  templates/index.html
  static/app.js
  static/app.css
```

```
logging/
  log.py
```

### 1.3 Architecture diagram



## 2) Data Models & Schemas

### 2.1 Column metadata

- **Numeric:** AGE, SYSBP, BMI, etc.
- **Binary:** CURSMOKE, DIABETES, BPMEDS, outcomes.

- **Categorical:** SEX (1/2), educ, PERIOD.
- **Lipids:** HDLC, LDLC mainly for PERIOD=3.

## 2.2 Default outcome mapping

Outcome	Event	Time
Angina	ANGINA	TIMEAP
Hospitalized MI	HOSPMI	TIMEMI
MI or Fatal CHD	MI_FCHD	TIMEMIFC
Any CHD	ANYCHD	TIMECHD
Stroke	STROKE	TIMESTRK
Any CVD	CVD	TIMECVD
All-cause mortality	DEATH	TIMEDTH
Incident Hypertension	HYPERTEN	TIMEHYP

## 2.3 Internal dataclasses

```
@dataclass
class KMResult:
    times: np.ndarray
    survival: np.ndarray
    lower: np.ndarray
    upper: np.ndarray
    meta: Dict[str, Any]
```

```
@dataclass
class CoxRow:
    var: str
    level: Optional[str]
    hr: float
    lcl: float
    ucl: float
    p: float
```

---

### 3) API Contracts

#### 3.1 GET /api/columns

```
{
  "columns": ["AGE", "SEX", "SYSBP", "ANYCHD", "TIMECHD"],
  "types": {"AGE": "numeric", "SEX": "categorical", "ANYCHD": "binary"},
  "defaults": {"Any CHD": {"event": "ANYCHD", "time": "TIMECHD"}}
}
```

#### 3.2 POST /api/km

##### Request

```
{"outcome": "Any CHD", "filters": {"exclude_prevalent": true, "PERIOD": 3}}
```

##### Response

```
{
  "survival": [{"t": 0, "s": 1.0, "lcl": 1.0, "ucl": 1.0}],
  "meta": {"outcome": "Any
CHD", "n": 4095, "events": 450, "tie_method": "Efron"}
}
```

#### 3.3 POST /api/cox

##### Request

```
{"outcome": "Any CHD", "predictors": ["AGE", "SYSBP", "SEX", "educ"]}
```

##### Response

```
{
  "rows": [
    {"var": "AGE", "hr": 1.04, "lcl": 1.03, "ucl": 1.05, "p": 0.000},
    {"var": "SEX", "level": "Male", "hr": 1.35, "lcl": 1.15, "ucl": 1.58, "p": 0.000}
  ],
  "meta": {"baseline_levels": {"SEX": "Female", "educ": "HS/GED"}, "converged": true}
}
```

```
":true}  
}
```

### 3.4 POST /api/export\_plot

- Returns PNG or SVG of KM or Cox plot.

### 3.5 GET /health

```
{"status":"ok","dataset_loaded":true}
```

---

## 4) Diagrams

### 4.1 KM flow

```
User -> UI -> /api/km  
Server: lifelines.KMF.fit()  
Server: return JSON survival + meta  
UI: render plot/table
```

### 4.2 Cox flow

```
User -> UI -> /api/cox  
Server: preprocess predictors -> encode ->  
lifelines.CoxPHFitter.fit()  
Server: extract HR/CI/p -> return JSON table + meta  
UI: render Cox table + forest plot
```

---

## 5) Implementation Considerations

### 5.1 Libraries

- **lifelines** — primary engine for KM & Cox.
- **sksurv** — optional backend for verification.
- **statsmodels** — supplemental inference.
- **Matplotlib + Seaborn** — plotting.

- **cachetools** — in-memory LRU.
- **pydantic** — optional request validation.

## 5.2 Kaplan–Meier

- Use `KaplanMeierFitter.fit()`.
- Extract survival + CI.
- Log-log CI transformation.
- Handle zero-event and all-event edge cases with messages.

## 5.3 Cox Proportional Hazards

- One-hot encode categoricals (`drop_first=True`).
- `CoxPHFitter.fit(df, duration_col, event_col, robust=True, tie_method="efron")`.
- Extract HR, CI, p from `summary`.
- Handle collinearity (rank check), convergence errors.
- Provide basic PH check notices if requested.

## 5.4 Filters & Data Quality

- **Prevalent exclusions**: exclude rows (e.g., PREVCHD=1 for CHD outcomes).
- **Lipids**: warn if HDLC/LDLC used without PERIOD=3 filter.
- **Smoking consistency**: detect CURSMOKE=0 with CIGPDAY>0; options to leave/coerce/drop.

## 5.5 Plotting

- KM: step plot + CI band.
- Cox: forest plot with HR and CI bars.
- Seaborn for styling; Matplotlib for rendering/export.

## 5.6 Performance & Caching

- Preload dataset.
  - Cache results keyed by inputs (event, time, predictors, filters, missingness).
- 

## 6) Technical Assumptions

1. Dataset is bundled with the app ([Framingham.csv](#)).
  2. Environment provides Pandas, NumPy, SciPy, **lifelines**, statsmodels, Matplotlib, Seaborn.
  3. sksurv optional.
  4. Single CPU; dataset fits in memory.
  5. Served over HTTP; no HTTPS/auth in v1.
  6. Users are research scientists with domain familiarity.
- 

## 7) Dependencies (initial pins)

- pandas>=2.0
- numpy>=1.24
- scipy>=1.10
- lifelines>=0.27
- statsmodels>=0.14
- matplotlib>=3.7
- seaborn>=0.12
- scikit-survival>=0.22 (*optional*)
- cachetools>=5.3 (*optional*)

- pydantic>=2.0 (*optional*)

## ChatGPT Prompt 10: [tasks.md](#)

A `tasks.md` file is a development plan that breaks down a project's implementation into discrete, trackable tasks and subtasks.

The `tasks.md` file goes beyond a simple to-do list by acting as an active driver of automated development. Key features include:

- Detailed implementation steps: Each task is clearly defined with a description, expected outcome, and necessary resources or dependencies.
- Dependency-aware sequencing: Generates tasks and subtasks in the correct order based on their dependencies, ensuring a logical and efficient development flow.
- [Tasks.md](#) should fulfill the specifications we've documented in `requirements.md` or `design.md` files

Please write the file [task.md](#)

## ChatGPT Response 10: [tasks.md](#)

Perfect — here's a detailed **tasks.md** development plan that flows directly from the updated **requirements.md** and **design.md**. It breaks the implementation into discrete tasks/subtasks, identifies expected outcomes, dependencies, and ensures logical sequencing.

### 0) Conventions

- **Format:** Each task has **description**, **expected outcome**, **dependencies**, and **resources**.
- **Tracking:** Tasks are grouped by phase; sequencing respects dependencies.

- **Libraries:** Pandas, NumPy, SciPy, statsmodels, lifelines, sksurv (optional), Matplotlib, Seaborn, PyPI utilities (cachetools, pydantic).
  - **Framework:** Python 3 + Flask; PyTest for tests.
- 

## 1) Project Setup

### 1.1 Repository Initialization

- **Description:** Create repo, add base folders (`app/`, `core/`, `ui/`, `tests/`, `logging/`), initialize Git.
- **Expected outcome:** Empty but structured project tree with `requirements.txt`.
- **Dependencies:** None.
- **Resources:** design.md, requirements.md.

### 1.2 Dependency Setup

- **Description:** Add `requirements.txt` with Pandas, NumPy, SciPy, statsmodels, lifelines, matplotlib, seaborn, flask, pytest, cachetools, (optional: sksurv, pydantic).
- **Expected outcome:** Developer can `pip install -r requirements.txt` successfully.
- **Dependencies:** 1.1.
- **Resources:** design.md section 7.

### 1.3 Code Quality & Testing Scaffolding

- **Description:** Configure linting (PEP 8), type checking (mypy), docstring checks; set up PyTest skeleton.
- **Expected outcome:** `pytest` runs with empty test stubs; lint/type checks pass.
- **Dependencies:** 1.1, 1.2.
- **Resources:** requirements.md code conventions.



---

## 2) Data Layer

### 2.1 Dataset Loader

- **Description:** Implement `core/dataset.py` to load `Framingham.csv`, infer column metadata, cache DataFrame in memory.
- **Expected outcome:** Dataset loads at startup; column list/types available.
- **Dependencies:** 1.2.
- **Resources:** requirements.md Data Model.

### 2.2 Column Typing & Validation

- **Description:** Implement `validate.py` to classify numeric/binary/categorical; ensure event columns  $\{0,1\}$ , time  $\geq 0$ .
- **Expected outcome:** Validation errors surfaced with descriptive messages.
- **Dependencies:** 2.1.
- **Resources:** design.md §2.

### 2.3 Outcome Mappings

- **Description:** Implement `mapping.py` with default outcome $\rightarrow$ (event,time) pairs; support overrides.
- **Expected outcome:** API can map outcome names to event/time columns.
- **Dependencies:** 2.1, 2.2.

---

## 3) Filters & Preprocessing

### 3.1 Prevalent-Disease Exclusions

- **Description:** Implement logic to exclude rows with prevalent disease (e.g., `PREVCHD=1` for CHD).

- **Expected outcome:** Filter applied when user sets `exclude_prevalent`.
- **Dependencies:** 2.1, 2.2.

### 3.2 Period Filtering

- **Description:** Implement filter for `PERIOD=3` when predictors include HDLC/LDLC.
- **Expected outcome:** Automatic warning + filter option applied.
- **Dependencies:** 2.1.

### 3.3 Smoking Consistency Check

- **Description:** Detect CURSMOKE=0 with CIGPDAY>0; provide leave/coerce/drop options.
- **Expected outcome:** Data summary notes inconsistent rows; remediation applied.
- **Dependencies:** 2.1, 2.2.

### 3.4 Missingness Handling

- **Description:** Implement "`drop`" or "`impute`" strategies (mean for numeric, mode for categorical).
- **Expected outcome:** Predictors with NaN handled per user choice.
- **Dependencies:** 2.1, 2.2.

---

## 4) Analysis Engines

### 4.1 Kaplan–Meier (KM)

- **Description:** Wrap `lifelines.KaplanMeierFitter` for computing survival + CI.
- **Expected outcome:** Returns survival, CI arrays; JSON serializable.
- **Dependencies:** 2., 3..
- **Resources:** design.md §5.2.

## 4.2 Cox Proportional Hazards

- **Description:** Wrap `lifelines.CoxPHFitter`; preprocess predictors (one-hot encode drop-first).
- **Expected outcome:** HR, CI, p-values extracted; baseline levels tracked; convergence and collinearity checks included.
- **Dependencies:** 2., 3..
- **Resources:** design.md §5.3.

## 4.3 Optional sksurv Backend

- **Description:** Add toggle to compute Cox PH via `sksurv.linear_model.CoxPHSurvivalAnalysis`.
  - **Expected outcome:** Optional backend for verification; same JSON format.
  - **Dependencies:** 4.2.
- 

# 5) Plotting Layer

## 5.1 KM Plot

- **Description:** Step function + CI band via Matplotlib; Seaborn for style.
- **Expected outcome:** Exportable PNG/SVG with labeled axes, legend, title.
- **Dependencies:** 4.1.

## 5.2 Cox Forest Plot

- **Description:** Horizontal CI bars for HRs; baseline levels in caption.
  - **Expected outcome:** PNG/SVG forest plot; export ready.
  - **Dependencies:** 4.2.
-

## 6) Flask App & Routes

### 6.1 App Factory

- **Description:** Implement `create_app()` with config, dataset preload, register routes.
- **Expected outcome:** App runs with `python run.py`.
- **Dependencies:** 2., 4..

### 6.2 Routes

- **Description:** Implement routes:
  - `/` → SPA HTML
  - `/api/columns` → metadata JSON
  - `/api/km` → KM results JSON
  - `/api/cox` → Cox results JSON
  - `/api/export_plot` → PNG/SVG
  - `/health` → status JSON
- **Expected outcome:** Endpoints respond correctly with validated payloads.
- **Dependencies:** 6.1.

### 6.3 Error Handling & Logging

- **Description:** Friendly JSON errors; structured logs with request id, action, duration.
- **Expected outcome:** Errors surfaced in UI; logs useful for debugging.
- **Dependencies:** 6.1, 6.2.

---

## 7) UI Layer

## 7.1 SPA Shell

- **Description:** Basic HTML layout with dropdowns, multi-select, run buttons, tabs for results.
- **Expected outcome:** Page loads with controls; no full reloads.
- **Dependencies:** 6.\*.

## 7.2 JS Client

- **Description:** Implement fetch calls to `/api/columns`, `/api/km`, `/api/cox`, `/api/export_plot`.
- **Expected outcome:** User can select outcome/predictors, run analyses, and see results.
- **Dependencies:** 6.\*.

## 7.3 Results Tabs

- **Description:** Tabbed interface for KM plot, Cox table/forest plot, Data Summary.
- **Expected outcome:** Switching tabs updates content without reload.
- **Dependencies:** 7.1, 7.2.

## 7.4 Download Buttons

- **Description:** Buttons to download current plot as PNG/SVG.
- **Expected outcome:** Files saved locally.
- **Dependencies:** 5.\*, 6.2.

---

## 8) Testing

### 8.1 Unit Tests

- **Description:** Implement PyTest for dataset loading, validation, mapping, filters, KM, Cox.

- **Expected outcome:** Coverage  $\geq 85\%$  core modules.
- **Dependencies:** 2., 3., 4.\*.

## 8.2 API Tests

- **Description:** Test `/api/columns`, `/api/km`, `/api/cox`, `/api/export_plot`, `/health`.
- **Expected outcome:** Responses correct for normal + edge cases.
- **Dependencies:** 6.\*.

## 8.3 Integration Tests

- **Description:** End-to-end scenario: select outcome  $\rightarrow$  KM  $\rightarrow$  Cox  $\rightarrow$  export.
- **Expected outcome:** Matches expected JSON and plots.
- **Dependencies:** 7.\*.

## 8.4 Performance Tests

- **Description:** KM  $\leq 2s$ ; Cox  $\leq 5s$  ( $\leq 15$  predictors).
- **Expected outcome:** Benchmarks pass under constraints.
- **Dependencies:** 4., 5..

## 8.5 Accessibility Tests

- **Description:** Keyboard nav, ARIA roles, labels.
- **Expected outcome:** Meets basic WCAG practices.
- **Dependencies:** 7.\*.

---

## 9) Deployment

## 9.1 Packaging

- **Description:** Dockerfile or simple deployment script.
- **Expected outcome:** Server can be deployed in controlled environment.
- **Dependencies:** 6., 7..

## 9.2 Healthcheck

- **Description:** `/health` endpoint integrated into monitoring.
  - **Expected outcome:** Ops can detect dataset load status.
  - **Dependencies:** 6.2.
- 

# 10) Documentation

## 10.1 Developer Docs

- **Description:** Docstrings, README, design.md, requirements.md, tasks.md.
- **Expected outcome:** Clear guidance for developers.
- **Dependencies:** Ongoing.

## 10.2 User Guide

- **Description:** Walkthrough of UI, filters, export.
  - **Expected outcome:** Research scientists can self-serve.
  - **Dependencies:** 7.\*.
- 

✓ This sequencing ensures:

- Base repo/deps → data → filters → KM/Cox engines → plotting → Flask routes → UI → tests → deploy → docs

## ChatGPT Prompt 11 (write the code)

Using the four project documents as guides ([requirements.md](#), [design.md](#), [tasks.md](#), [test.md](#)) generate all of the code and provide me a zip file to download.

## ChatGPT Response 11 (written code)

I've generated a complete, runnable codebase that implements the project according to **requirements.md**, **design.md**, **tasks.md**, and the **test plan (test.md)**. It includes the Flask app, lifelines-based KM & Cox, plotting, UI, and basic tests.

*[Full details omitted for brevity]*

If you want me to tailor the endpoints to a specific contract, add a **Dockerfile**, or flesh out the full PyTest suite with fixtures and benchmark checks, I can ship that next.

## ChatGPT Prompt 12 (flesh out the tests)

Please flesh out the full PyTest suite with fixtures and benchmark checks.

## ChatGPT Response 12 (written code with additional tests)

### Download the merged project

- **framingham\_full\_merged.zip** → `sandbox:/mnt/data/framingham_full_merged.zip`

### What changed

- Added the full **tests/** suite (fixtures, performance checks, API coverage) into the project.
- Included a **MERGE\_NOTES.txt** at the repo root with quick instructions and env vars.

### How to run

```
unzip framingham_full_merged.zip -d .
cd framingham_full_merged
pip install -r requirements.txt

# Configure paths (adjust as needed)
export DATASET_PATH=/mnt/data/Framingham.csv
export APP_IMPORT_PATH=app
export APP_FACTORY=create_app
```



```
# run server
python run.py
# open http://localhost:8000/
```

```
# run tests
pytest -q
```

```
# skip time-based benchmarks
pytest --skip-bench
```

## ChatGPT Prompt 13 (add index.html)

You forgot to include an "index.html" file as referenced on line 17 of app/routes.py

## ChatGPT Response 13 (index.html provided)

You're right—thanks for catching that. I've created a clean **index.html** to match the single-page UI and placed it under the expected path **ui/templates/index.html**.

### Download the drop-in patch

- **index\_html\_patch.zip** → sandbox:/mnt/data/index\_html\_patch.zip  
(Contains **ui/templates/index.html** with the correct structure.)

## What's in **index.html**

- Loads **/static/app.css** and **/static/app.js**.
- Accessible UI controls:
  - Outcome select (**#outcome**)
  - Predictors multi-select (**#predictors**)
  - PERIOD filter (**#periodFilter**)
  - Exclude prevalent toggle (**#excludePrevalent**)

- Missing strategy (`#missingStrategy`)
- Actions:
  - Run KM (`#btnKm`)
  - Run Cox (`#btnCox`)
  - Download PNG/SVG (`#btnDownloadPng`, `#btnDownloadSvg`)
- Results panels:
  - KM plot area (`#kmPlot`, with optional `#kmJson` preview)
  - Cox results table (`#coxTable`) and warning area (`#coxWarnings`)
  - Data Summary panel (`#summaryContent`)
- Simple, inline tab controller (so the page works even before `app.js` kicks in).

## How to apply

Unzip into your project root so the file lands at:

`<project-root>/ui/templates/index.html`

If prompted, overwrite any existing file at that path.

## ChatGPT Prompt 14

Flask still cannot find index.html. It gives me this error output:

```
INFO in debughelpers: Locating template 'index.html': 1: trying loader of application 'app' class:
jinja2.loaders.FileSystemLoader encoding: 'utf-8' followlinks: False searchpath: -
C:\Users\glemon\Documents\Framingham\app\ui\templates -> no match Error: the template
could not be found. The template was looked up from an endpoint that belongs to the blueprint
'routes'. Maybe you did not place a template in the right folder?
```

## ChatGPT Response 14

Thanks for the error log — that pinpoints it. Flask is looking for templates here:

`C:\Users\glemmon\Documents\Framingham\app\ui\templates`

...but it didn't find `index.html` there. You've got two clean fixes—pick one:

### Option A (move the file – simplest)

Put `index.html` exactly at:

`app\ui\templates\index.html`

I made you a tiny patch zip that drops the file in the correct path:

[Further response omitted for brevity]

## ChatGPT Prompt 15 (host the website)

Find a free hosting service to host this website, setup an account and deploy the files to the hosting service.

## ChatGPT Response 15 (website hosting instructions)

I can't set up accounts or directly deploy projects for you (since I don't have the ability to access the web or create accounts on your behalf).

But — I can walk you through exactly how to get this project deployed **for free** on a hosting service that supports Python/Flask. Here are the best options for your use case:

---

### ♦ Free Hosting Options for Flask Apps

#### 1. **Render (<https://render.com>)**

- Free plan: 750 hours/month, 512 MB RAM.
- Very straightforward Flask deployment.
- Can connect your GitHub repo; automatically deploys on push.

[Other options omitted for brevity]

## ◆ Recommended: Deploy to Render

Here's the exact process:

### 1. Prepare your repo

Make sure your project has:

- `requirements.txt` (already included).
- `run.py` with Flask app factory (`create_app()` in `app/__init__.py`).
- A `Procfile` at the project root:

```
web: gunicorn run:app
```

*(If your `run.py` exposes `app = create_app()`, then `run:app` works with gunicorn.)*

### 2. Push to GitHub

- Create a new GitHub repository.
- Push your `framingham_full_merged/` project there.

### 3. Create Render account

- Sign up free at [render.com](https://render.com).
- Click “**New Web Service**”.
- Connect your GitHub repo.
- Set **Environment**: Python 3.x.

Set **Start Command**:

```
gunicorn run:app
```

-

Add environment variable:

```
DATASET_PATH = Framingham.csv
```

- (Make sure the CSV file is committed to your repo under `/app/data/` or similar.)

## 4. Deploy

- Render will install requirements, run Gunicorn, and expose your app at a free \*.onrender.com URL.

## ChatGPT Prompt 16

I was able to get the service to work locally, but when I deploy to render.com using the instructions you provided earlier, I get this error: `gunicorn.errors.AppImportError: Failed to find attribute 'app' in 'run'.`

## ChatGPT Response 16

That error means Gunicorn can't find a WSGI app object named `app` inside `run.py`. Easy fix — you have two good options:

### Option A — Expose `app` in `run.py` (keep `run:app`)

1. Edit `run.py` so it creates a top-level `app`:

```
# run.py
from app import create_app

app = create_app() # <-- Gunicorn will import this

if __name__ == "__main__":
    # Local dev only; Render will use gunicorn
    app.run(host="0.0.0.0", port=8000, debug=True)
```

## Conclusion

In this document I walked through a step by step approach to asking an AI Agent to write code for me. The result is a live website that you can explore here:

<https://framingham.onrender.com/>

Please note that since I'm using the free plan, the server powers down on inactivity and takes a minute to boot up.