

DESIGN REPORT

CLINI: A REUSABLE ON-PREMISE LLM PLATFORM FOR HEALTHCARE RESEARCH

Harry van Dijk – s3205703
Jens Taken – s3136027
Juras Mačijauskas – s3151913
Marlon Nelson – s3185702
Ou-Xiang Chen – s2980045
Sander Bertram – s3193004

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
in collaboration with Deventer Hospital

Supervised by:
Nacir Bouali

NOVEMBER 13, 2026

Abstract

This report is written as a result of the Design Project at the University of Twente. A system named CLINI has been developed for Deventer Hospital. The motivation of developing this application was that hospitals increasingly want to leverage Large Language Models (LLMs) for flexible and high impact use cases. In this case more specifically, to extract structured information from unstructured Electronic Patient Records (EPRs). In the current course of actions, the identification of important data in patient's history requires manual review by researchers, which is very time consuming and also error-prone. Due to the highly sensitive nature of medical data, the use of online LLMs is unacceptable. This restricts researchers to rely on on-premise GPU LLM infrastructure. The use of these systems requires technical knowledge that most researchers do not have. In the end, the system should facilitate and empower research with the power of LLM, without requiring additional technical knowledge. The knowledge that is required should be minimal and with the click of a few buttons, a researcher should be able to simply do their job. With this, a powerful LLM can be used on-premise. The application could be distributed or shared with other hospitals. For privacy reasons, the project was executed and tested locally. For testing and general project execution, a realistic replica of patient data was used.

Contents

1 Introduction **6**
- 1.1 The Company and Domain Analysis . . . 6
  - 1.1.1 Motivation . . . 6
- 1.2 The Healthcare Data Domain . . . 6
  - 1.2.1 EPR and the unstructured text problem . . . 6
  - 1.2.2 Characteristics of clinical text . . . 6
  - 1.2.3 Reuse of EPR data . . . 7
  - 1.2.4 Legal and ethical aspect . . . 7
- 1.3 LLMs for Clinical Information Extraction . . . 8
  - 1.3.1 LLMs in medicine . . . 8
  - 1.3.2 Row-by-row extraction with prompt templates . . . 8
  - 1.3.3 Reliability and validation . . . 8
  - 1.3.4 Running on local hardware . . . 8
- 1.4 Stakeholders . . . 9
- 1.5 Software Environment and Tool Choices . . . 9
  - 1.5.1 Python and asyncio . . . 9
  - 1.5.2 FastAPI and SQLite . . . 9
  - 1.5.3 Ollama . . . 9
  - 1.5.4 Other models to compare . . . 10
- 1.6 Current procedure . . . 11
- 1.7 Conclusions . . . 12

2 System Specification and Proposal **14**
- 2.1 Constraints . . . 14
  - 2.1.1 Privacy, legal and ethical constraints . . . 14
  - 2.1.2 Hardware and infrastructure constraints . . . 14
  - 2.1.3 Model and data constraints . . . 14
  - 2.1.4 Users and project constraints . . . 15
- 2.2 How constraints shaped the decisions . . . 15
- 2.3 Requirements . . . 15
- 2.4 Deployment and Demonstration . . . 17
- 2.5 Risk analysis . . . 17
- 2.6 Planning and milestones . . . 18
- 2.7 Requirement traceability and status . . . 19

3 General System Design **20**
- 3.1 Architecture Overview . . . 20
  - 3.1.1 Deployment view . . . 21
- 3.2 Design Decisions . . . 21
  - 3.2.1 Backend . . . 21
  - 3.2.2 Frontend . . . 22
- 3.3 Backend design . . . 22
  - 3.3.1 Domain model . . . 22
  - 3.3.2 Authentication . . . 23
  - 3.3.3 API . . . 24
  - 3.3.4 Run-time validation of input . . . 25
  - 3.3.5 Scheduler . . . 26
  - 3.3.6 Scheduler and worker design . . . 28
  - 3.3.7 LLM Layer . . . 29

3.4 Frontend Design . . . 31
3.4.1 Frontend Architecture . . . 31
3.4.2 Task page . . . 32
3.4.3 Project Page . . . 33
3.4.4 Job Page . . . 33
3.4.5 Prompt Templates Page . . . 34

4 Detailed Implementation **35**
4.1 Backend . . . 35
4.1.1 Authentication and authorization . . . 35
4.1.2 Database and service boundary . . . 35
4.1.3 LLM client and prompt construction . . . 35
4.1.4 Boundary validation and typed outputs . . . 36
4.1.5 Scheduler, batching and concurrency . . . 36
4.1.6 Pause, resume, cancel, progress and logging . . . 36
4.1.7 Deployment . . . 37
4.2 Frontend . . . 37
4.2.1 Frontend implementation choices . . . 37
4.2.2 Login/registration page . . . 38
4.2.3 Task overview . . . 39
4.2.4 Projecten . . . 40
4.2.5 Sjablonen . . . 45
4.2.6 Help . . . 47
4.3 User and developer guide . . . 47

5 Testing **48**
5.1 Test Plan . . . 48
5.2 Unit and integration testing . . . 48
5.3 User acceptance testing . . . 49
5.3.1 Testing plan . . . 49
5.3.2 User acceptance and demonstrations results . . . 50
5.3.3 Evaluation of results and advise . . . 51
5.4 Testing Mistral against other models . . . 51
5.4.1 Evaluation . . . 53
5.5 Extra results Mistral . . . 54
5.5.1 Local inference throughput and capacity . . . 54
5.5.2 Confirmation of the memory estimate . . . 54
5.6 Large limitations to the evaluation . . . 54

6 Discussion **55**
6.1 General view . . . 55
6.2 What worked . . . 55
6.3 Limitations of the research and evaluation . . . 55
6.4 Technical and methodological lessons . . . 56
6.5 Responsible use and handover . . . 56
6.6 Future work . . . 56

7 Conclusion **58**

8 Project Management **59**
8.1 Justification of approach . . . 59
8.2 GitLab planning structure . . . 59

8.3 GitLab labels and epics . . . 60
8.4 Sprint evaluation and adaptation . . . 61
8.5 Risk and scope management . . . 62
8.6 Definition of Done and quality control . . . 62

9 Contributions **63**

10 Reflection **65**
10.1 Process . . . 65
10.2 What would be done differently . . . 65

A Terminology **68**

B AI Statement **69**

C Clinical system instruction **70**
C.1 Machine-readable JSON instructions . . . 70

D Algorithms **72**
D.1 Dependency-aware batch scheduling . . . 72
D.2 Output validation and retry . . . 72
D.3 Simple pause and resume . . . 73

E Weekly sprint summary **74**

F API Endpoints **76**

G Client meeting notes **77**

H Supervisor meeting notes **78**

I User test plan **79**
I.0.1 Instructie voor de deelnemer . . . 79
I.0.2 Deel 1 – Taken . . . 79
I.0.3 Deel 2 – Vragen achteraf . . . 79

J User test answers **81**
J.0.1 Algemene ervaring . . . 81
J.0.2 Projecten en prompts . . . 81
J.0.3 Bestanden, analyses en resultaten . . . 82
J.0.4 Verbeterpunten . . . 82
J.0.5 Afsluiting . . . 83

K Risk assessment (P×I) **84**

List of Figures

1 Flowchart of the current data-request, prompt, execution, and review procedure. . 11
2 General system overview and data flow. 21
3 Domain and persistence classes used by the backend. 23
4 Sequence of registration, login, and token refresh. 24
5 Project and job management sequence using the backend objects. 26
6 Scheduler and worker interaction derived from the backend classes. 29
7 Original sketch of task overview 33
8 Original new prompt page sketch 34
9 Login page 38
10 Registration page 39
11 Task overview 40
12 Projects overview page 41
13 Project view 42
14 New prompt popup 43
15 Earlier output columns selection with filters 44
16 Output viewer 45
17 Prompt template page 46
18 Add new prompt template 46
19 Help page 47
20 Accuracy vs temperature for the testing models. 52
21 Accuracy heatmap by model and temperature. 52
22 Invalid or missing output rate by model and temperature. 53
23 GitLab issue board used for day-to-day planning, assignment, and tracking of project work. 60
24 GitLab epic board showing the large project themes and their associated work items. 60
25 The GitLab label scheme used to classify and track issues. 61

List of Tables

1 Users and interested parties. . . . 9
2 Some other models including measured/known values . . . 10
3 Prioritised functional requirements. . . . 16
4 Prioritised non-functional requirements. . . . 17
5 Project risks, mitigation measures, and observed outcomes. . . . 18
6 Traceability from requirements to design and verification evidence. . . . 19
7 Project lifecycle states. . . . 27
8 VRAM and KV use per amount of workers for `mistral-small:24b`. . . . 31
9 Python test files, covered subsystems, coverage, and results. . . . 49
10 Observed Mistral capacity test and corresponding memory estimate. . . . 54
11 Contributions per team member. . . . 63
12 Endpoints of the production API. . . . 76
13 P×I risk assessment. . . . 84
14 Risk matrix (probability and impact) . . . 84

1 Introduction

1.1 The Company and Domain Analysis

The Deventer Hospital (DZ) is a general hospital located in Deventer. It provides medical care to patients in the surrounding region. Like other hospitals in the Netherlands, it generates a large amount of Electronic Patient Records (EPRs), many of which consist of free text such as doctor reports and telephone consultations. These records contain valuable information for medical research. The project was carried out with the DZ data department, which had already built a small Python application on a local Linux machine to experiment with LLMs. The data department supports researchers who are mostly medical students and physicians. Until now, they often spent weeks manually reading patient data and collecting information from EPRs.

1.1.1 Motivation

Currently, tasks like identifying patient history often require manual review of records by students or researchers, which is time-consuming and error-prone and due to the highly sensitive nature of medical data, the use of external cloud-based LLM services such as ChatGPT is not acceptable. Hospitals therefore increasingly rely on on-premise GPU infrastructure.

Using these systems, however, requires technical and coding knowledge that most medical researchers do not have. To facilitate research, researchers should be empowered with the power of LLMs, without the need for extra technical knowledge or a steep learning curve.

This report follows the path from the domain problem to the system designed to address it. It first explains the healthcare data and LLM context, identifies the stakeholders and current workflow, and derives the constraints and requirements. It then presents the architecture, implementation, and design rationale, followed by testing and model evaluation. Finally, the discussion reflects on what worked, what remained constrained, and what would change in a less restricted deployment.

1.2 The Healthcare Data Domain

1.2.1 EPR and the unstructured text problem

Hospitals record any kind of patient care in an EPR. An EPR combines structured data (any kind of coded diagnoses, measurements, etc) with unstructured data, mainly text and consultation reports. The structured fields are ready for use in research, but they are sometimes time consuming to fill in or are even left incomplete. Pure text is quick and mostly expressive for the clinician, so a large share of clinically relevant information ends up in the notes. Rosenbloom et al. describe this as a fundamental tension between structured and flexible documentation [1] that could maybe be simplified.

This conclusion matches exactly with what the client reported during the first meetings: questions asked on smoking, diseases, or medication use do exist in an EPR but are rarely filled in completely, so physicians describe the patient in text instead. The information researchers need (for example, BMI and general conclusions) is therefore present in the record but cannot easily be found by a researcher. In this case it is interesting to investigate whether an LLM can simplify, or even speed up this process.

1.2.2 Characteristics of clinical text

Clinical text of course has its characteristics. These differ from general language, which makes automatic processing almost impossible. [2] What is often seen in clinical text is typically [3]:

- **Abbreviations and loads of jargon** Use of local abbreviations and medical terminology.
- **Grammar and typos** Notes are written under time pressure. The client gave the example of mistyped blood pressure values.
- **Context dependence** A measurement can be an actual clinic value, a quick measurement, or even a reference. Together with the written text, these measurements get their meaning. A condition can be affirmed, negated, hypothetical or about a family member and not about the patient themselves.
- **Language** The records at DZ are all in Dutch, for which there are fewer testing resources and structuring implementations than for English.

Given these, research shows that extracting information from text with natural language processing (NLP) has also been an active area, but that traditional boolean and supervised systems require substantial effort to build and adapt. [4].

1.2.3 Reuse of EPR data

Using clinical data that were routinely collected for research (secondary use) comes with some data quality issues. The purpose of an EPR is to document data for care, not research, and may be incomplete, inconsistent, or biased [5, 6]. In practice, researchers at DZ spend weeks manually reading records to collect the variables they need specifically. And this process has to repeat when a new variable turns out to be relevant. The workflow of a researcher is basically a loop: research a variable, extract it, check the result, and refine the definition.

1.2.4 Legal and ethical aspect

Health data falls under a special category of personal data under the General Data Protection Regulation (GDPR) [7], and processing it is subject to very strict conditions. In the Netherlands, hospitals also must follow the NEN 7510 for healthcare [8] information security standard. These create some consequences that are relevant to this project:

- Sending patient data to any cloud service (In this case a cloud LLM) is not acceptable for DZ. All processing must happen **on premise**.
- The source data cannot be persisted on the platform, since the program might be shared between hospitals. Only prompt templates and processed results can be stored. Identifiers may be strings or other project-defined values; the hospital is responsible for choosing identifiers that do not unnecessarily reveal a patient when results are shared.
- De-identification of text is a common procedure to share and reuse data. For Dutch clinical text, the DEDUCE method is an accepted approach [9].
- The data department of the DZ is legally and ethically responsible for the sharing of data.

Dennstädt et al. [10] discuss the common tradeoff between control, collaboration, cost, and security when implementing LLMs in a hospital. Moreover, it is argued that locally hosted open models are an important option for using this sensitive data.

1.3 LLMs for Clinical Information Extraction

1.3.1 LLMs in medicine

LLMs have shown very solid performance on medical knowledge data extraction, and on multiple clinical language tasks [11, 12]. For basic information extraction, Agrawal et al. showed that LLMs can be information extractors, performing tasks such as medication or abbreviation extraction with only a few examples instead of a large annotated training set [13]. The prompt design strongly influences the result, as shown for entity recognition in a clinic by Hu et al. [14]. LLMs trained on clinical notes have also been used for other prediction tasks [16]. Moreover, Wiest et al. showed that LLMs run locally can convert unstructured clinical text into structured data, also preserving privacy [17].

1.3.2 Row-by-row extraction with prompt templates

For the desired workflow, a dataset is a CSV table with one row per record (e.g. a doctors report). A *prompt template* with placeholders is filled with the content of one or more columns of each row, then sent to the LLM, and the answer is parsed into a typed result. This could be a score, number, category, boolean value, or text. A typical example is extracting blood pressure from notes, classifying whether a telephone consultation meets the billing criteria, and judging risk in incident reports. Long prompts that ask for many things at once can confuse the model, so the client splits the tasks into a batch of rows, which result in shorter prompts. This structure also allows for sequential prompting, which is reusing the output of a prompt as input for a new prompt.

1.3.3 Reliability and validation

LLMs also produce incorrect output: this can sound truthful but still be false. This is a phenomenon known as hallucination [18]. Current LLMs have their limitations on this because a model is never 100% accurate [19]. Very long inputs can also reduce accuracy, because for example some models use information in the middle of long contexts less reliably [20]. For a research tool this means most of the output must be validated. Strategies are:

- manually checking a sample of rows,
- annotating a subset of rows and use that as golden dataset,
- using an LLM as a judge procedure [21], in which another model judges the output; this is an evaluation method, not agentic AI,
- storing a validation column that the system can compare against manually.

To ensure the output is in the correct format, the implementation uses schema validation with typed output columns, allowed category values, and explicit handling of missing values through Pydantic model checking [29].

1.3.4 Running on local hardware

Running a model locally introduces its limits, namely GPU memory. Memory must hold the model weights and, for every concurrent request, the key-value (KV) cache that stores context. KV cache memory grows with the context length and the number of parallel requests. Managing this is a central problem in the management of an LLM [22]. This clearly is a constraint: the maximum context per user and the number of parallel sequences must fit spaciously in the remaining GPU memory. Quantization of an LLM is an option, this reduces weight memory by storing weights at lower precision with limited loss of quality [24], most models that run locally are already quantized.

1.4 Stakeholders

Table 1 lists the users and interested parties, collectively called stakeholders. They are ordered according to how directly they are affected by or influence the system.

Users	Role	Interest
Medical researchers	Primary end users	Extract research results from text without programming; write, test and rewrite prompts; export results to CSV.
DZ BI department	Client and technical contact; current maintainers	A rework of the workflow with a maintainable, reusable platform.
DZ research department	Customer of the BI department	Faster data collection.
Hospital IT and information security	Infrastructure owners	The machine is isolated from the hospital network; no hospital single sign-on; compliance with security policy.
Data protection and ethics bodies	Oversight	Compliance with GDPR and ethical rules and no storage of source data.
Patients	Indirect stakeholders	Privacy of their records, correctness of research derived from them.
Other hospitals	Potential future users	Software that is reusable with different LLM endpoints.

Table 1: Users and interested parties.

The primary users are medical researchers. They are experts in medicine, are used to spreadsheet tools such as Excel, but have little programming knowledge. They typically work with CSV exports of up to roughly on average 10,000 rows. The system therefore has to hide some technical details behind a simple interface.

1.5 Software Environment and Tool Choices

1.5.1 Python and asyncio

The analysis scripts and the API are written in Python. The script waits for the model to give an output, which suits Python's `asyncio` library. In the current script a single event loop runs many requests concurrently [25].

1.5.2 FastAPI and SQLite

The architecture uses FastAPI for the local API and SQLite for application metadata. The reasons for these choices are discussed in Section 3.2; this subsection only establishes the technology baseline used by the architecture. SQLite stores users, project definitions, statuses, and processed results, while source CSV data remains outside the database.

A larger networked deployment with many simultaneous users could justify PostgreSQL, but that is outside the scope of this system. Source CSV data is not stored in SQLite because of the privacy constraint.

1.5.3 Ollama

Ollama is an application and runtime for running LLMs locally. With Ollama the model is exposed through an API [32]; it typically serves quantised models and uses the llama.cpp inference engine [33]. The deployed model is `mistral-small:24b`, using the Q4_K_M quantisation with approximately 14 GB of weights. The weights are loaded once and shared by parallel requests. Ollama

allocates a KV-cache context for each parallel request, so the configured context length and the number of parallel requests determine the additional GPU memory requirement.

For the deployment, concurrency is enabled explicitly with `OLLAMA_NUM_PARALLEL=4`;

1.5.4 Other models to compare

Because the platform should be reusable, other open-weight models are relevant too. Table 2 compares some model. They assume an RTX 6000 Ada, 4-bit quantisation, a 16-bit KV cache, a moderate prompt, and one active request; the actual values depend strongly on the quantisation, runtime, context length, and Ollama configuration.

Model	**Size**	**Max context (native)**	**4-bit weights**	**KV cache per 8k sequence**	**Tokens/s (measured/estimated)**	**Remarks**
Mistral Small 3 [35]	24B	32k	≈14 GB (Q4_K_M)	≈1.3 GB	≈40–60 (measured)	Current `mistral-small:24b` model at DZ; typical range on an RTX 6000 Ada.
Qwen3 [36]	32B (dense)	32k (128k with extension)	≈20 GB	≈2.0 GB	≈25–45 (rough estimate)	Multilingual; its "thinking" mode can add latency.
Gemma 3 [37]	27B	128k	≈17 GB	≈1.7 GB (rough estimate)	≈30–50 (rough estimate)	Multilingual with long context; these two values are estimates rather than project measurements.
Llama 3.3 [38]	70B	128k	≈40 GB	≈2.6 GB	≈8–20 (rough estimate)	The estimated weights plus reserve leave no safe worker under the conservative 48 GiB policy.

Table 2: Some other models including measured/known values

1.6 Current procedure

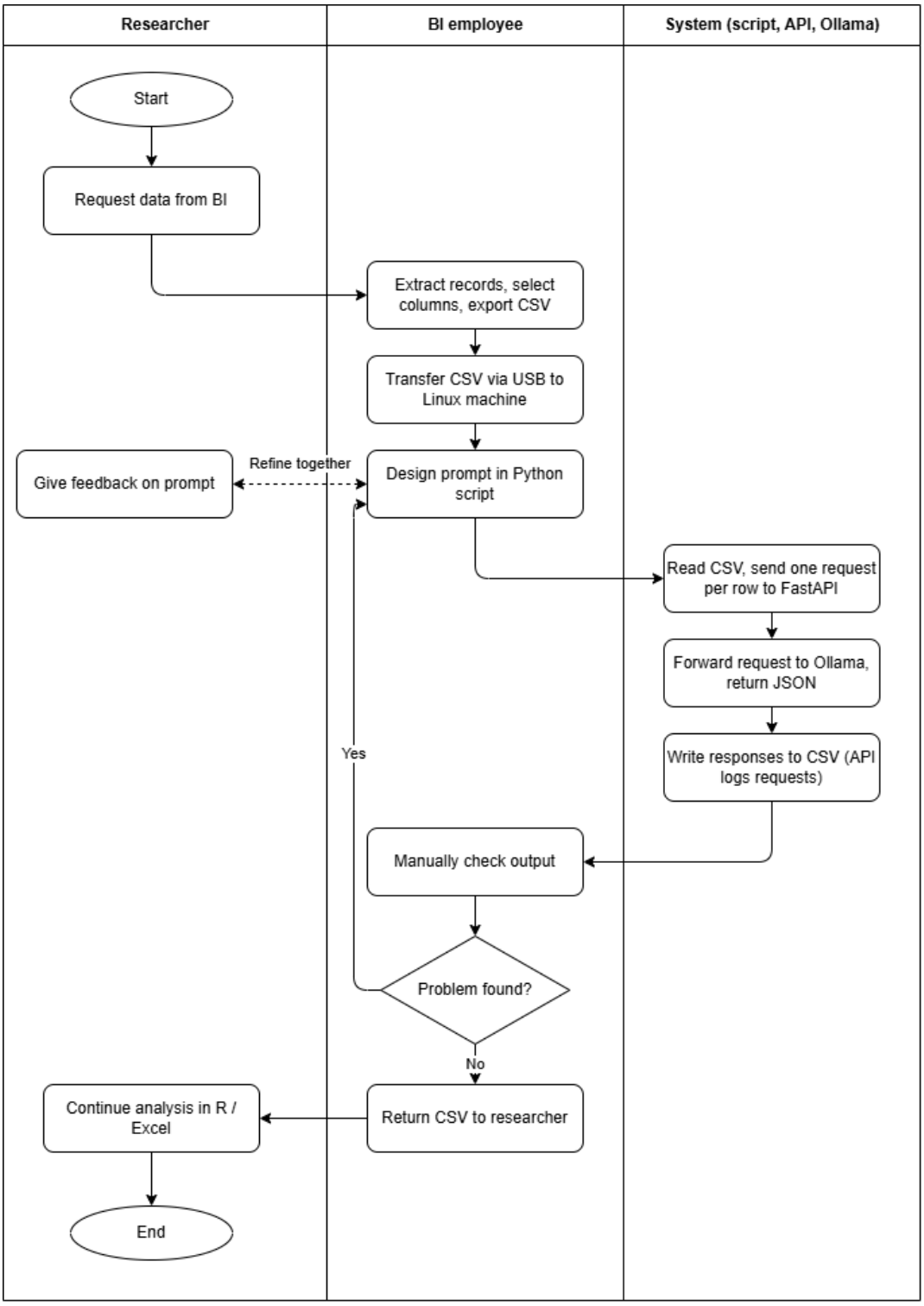


Figure 1: Flowchart of the current data-request, prompt, execution, and review procedure.

At the DZ, the BI department has already built a small script for the text analysis system. It is operated manually by the BI staff. Several employees have accounts on the machine (hardcoded in the setup scripts, since the hospital single sign-on cannot be used). The current procedure is as follows:

1. **Request and data preparation:** A CSV with data is requested from BI by a researcher. Records are extracted by BI from the hospital's datacenter. The relevant columns are selected and exported to CSV.
2. **Transfer:** The CSV is transferred via a USB stick to the Linux machine.
3. **Prompt design:** A BI employee writes the prompt, inserts it in the Python scripts, using shared helper functions that clean the text and fill a prompt template. Researchers are not involved at all in writing prompts, so BI and researchers have to repeatedly refine them.
4. **Execution:** The script reads the CSV and sends one request per row to the FastAPI service, using the four workers. The API forwards each request to the local Ollama server, which runs Mistral Small 24B and returns the output. The outputs are then stored as JSON.
5. **Parsing and saving:** The script reads the response and writes it to a CSV file. The API separately logs each request in this scenario.
6. **Review and delivery:** The CSV is then manually checked and returned to the researcher, who continues in R and Excel. If a problem is found, the process starts again from the prompt design onward.

The limitations in this course of actions are:

- There is no UI. Operating the system requires Python and Linux knowledge, so researchers depend on BI staff. The FastAPI Swagger is only a technical test page.
- A crash after hours of processing means starting over, and there is no pause or resume and almost no feedback to the user.
- There is no queue, so concurrent use by several users would interfere with each other.
- Only hardcoded CSV is analyzed per run, and column names and identifiers are tied to the specific use case.
- The API is tied to one model and endpoint.
- Users are hardcoded.
- Validation of the output (format) is manual, and error handling for invalid or missing output is limited.

1.7 Conclusions

From the analysis, the following conclusions are drawn for the design of the system.

1. Relevant clinical information is mostly in text, and manual extraction costs researchers plenty of time. Row-by-row extraction of LLM is a feasible approach. [13, 17].
2. The platform must run completely offline in the hospital. It cannot and may not persist source data. It needs its own login because the hospital single sign-on is unavailable.
3. The main issue is usability, there is no web interface for designing, testing and rerunning prompts on a selection of rows, and for viewing and exporting results, this is the core of the system.
4. Because the LLM's output formatting can be wrong [18, 19], the system needs typed output columns. No parsing and other error handling is included.

5. A persistent queue with a fair queuing system, pause, resume, progress, and recovery after errors should replace the manual worker loop, and gives several users safe access to a single GPU.
6. A single 48 GB GPU, the model size and the per-user KV cache limit the number of parallel workers and the context length. These limits were estimated and documented using [22].
7. The platform must not depend on one dataset, one model or one endpoint, so models and endpoints must be configurable. The candidate models should be compared on quality, speed, memory and KV-cache use.

2 System Specification and Proposal

The goal of this project is to design and prototype a system to process datasets with an on-premise LLM.

2.1 Constraints

The following constraints limit the design and implementation. They come from the meetings with the client (DZ BI department), the supervisor, and the analysis in the previous chapter.

2.1.1 Privacy, legal and ethical constraints

- **No storage of source data.** The platform may persist only prompt templates, job configurations and processed results. The source CSV must be uploaded again for each session and may be kept in the browser only temporarily.
- **Identifiers in results.** Results may be stored because they cannot be linked back to a patient, except through the row identifier. Identifiers must be left out if results are shared.
- **Responsibility.** The BI, data, and ethics department of the DZ are responsible for distributing and handling the data.
- **Development data.** Real patient records are not available to the team. Development and testing use public or artificial data with a realistic structure.
- **Logging policy.** Metric logs contain model name, temperature, token counts, durations, statuses, errors, and nonsensitive row counts. They do not contain prompt text, CSV cells, patient numbers, or raw model responses. A patient number may be used as a temporary row identifier during a local run, but it must be removed or pseudonymised before results are shared. Free-text outputs are treated as potentially sensitive and are therefore subject to the same researcher access controls as other results.

2.1.2 Hardware and infrastructure constraints

- **Single GPU.** The hospital machine is a Linux desktop with one NVIDIA Ada 6000 GPU 48 GB. It has no server configuration and cannot be moved to the data center.
- **Memory budget.** GPU memory must hold the model weights and the KV cache of every parallel request. The model size, context length per user and number of parallel sequences are therefore bounded. The client currently runs four parallel prompts; but beyond a specific amount of workers, processing spills to the CPU and slows down sharply.
- **Isolated machine.** The machine is not connected to the hospital network. Internet access is switched on only to download models. Data is transferred by USB stick. The hospital single sign-on cannot be used, so the platform needs its own login. There is no remote access, so the team cannot test remotely and the final software must be deployable by the hospital itself.
- **Expected load.** The client expects at most one concurrent user, but the design must allow several users to work without interfering with each other.

2.1.3 Model and data constraints

- **Model and runtime.** The client currently uses `mistral-small:24b` through Ollama, with Q4_K_M quantisation and approximately 14 GB of weights, and has tested it on Dutch text.

Their existing API errors when other models are used, so model independence has to be built in. alternative models are discussed on Chapter 5.4.

- **Dutch language.** Records are in Dutch, so any model considered must handle Dutch clinical text.
- **Input format.** Input is a CSV file with a selectable identifier column. Direct SQL access and changes to the hospital database are out of scope. Files can reach about 500,000 rows, so the interface must page the data. Browsers also limit the size of data held in memory (for Chrome roughly 5.4×10^8 bytes in a string), which constrains how large files can be handled on the client side.
- **Output reliability.** LLM output can be wrong. The platform provides typed output columns and error reporting, but full validation of results is limited to recommendations.

2.1.4 Users and project constraints

- **Nontechnical users.** Researchers know Excel or SPSS but are not experienced programmers. The system must be usable without deep technical knowledge.
- **Time.** The project ran for one quarter. The final poster was due on 4 November, and the final presentation with the client took place on the 29th October. This forced the scope to be prioritised using the MoSCoW ranking agreed with the client.

2.2 How constraints shaped the decisions

The constraints are not only a list of limitations; they explain why the final design looks the way it does. The isolated machine and privacy requirements led the project to keep inference local, avoid external identity providers, and upload the source CSV for each run instead of persisting it. The single 48 GB GPU led to a bounded asynchronous scheduler, configurable batch sizes, and conservative parallelism rather than unrestricted workers. The nontechnical user group led to a browser interface, progress reporting, typed outputs, and explicit error messages instead of a command-line workflow. Finally, the short project and the absence of real patient records meant that the project prioritised a reusable prototype and tested with realistic artificial data rather than attempting a clinical validation study.

If these constraints did not apply, approaches such as a hospital network deployment with central authentication, a managed database, direct integration with approved data sources, larger multi-GPU inference, and a broader validation study with de-identified clinical records exist that would not only increase scalability and validation strength, but they would also introduce additional security, operational, and governance responsibilities outside this project's scope.

2.3 Requirements

The requirements were defined together with the client and prioritised using the MoSCoW method: Must have, Should have, and Could have. Data is loaded from CSV files only. Direct hospital database integration, hospital single sign-on, clinical correctness guarantees, and unrestricted multi-GPU deployment were project constraints and scope exclusions, not additional requirements.

Priority	ID	Requirement	Justification
Must have	FR1.1	The system shall ensure correct CSV retrieval and parsing of data.	Data retrieved must be correct to have correct results.
Could have	FR1.2	The system shall be able to normalize categorical values.	Categorical values may have different representations across reports or data sources. Normalizing improves data consistency and reduces ambiguity during LLM analysis.
Should have	FR1.3	The system shall apply defined handling of missing/unknown values in the dataset.	To prevent errors from propagating through the system.
Must have	FR2.1	The system shall dynamically construct prompts based on the user's analysis configurations.	Researchers could have multiple (specific) questions they want an answer to.
Must have	FR2.2	The system shall support including one or more fields from each data row in a dynamically constructed prompt.	Some rows may introduce ambiguity or be completely useless.
Could have	FR2.3	The system shall allow additional options to be added on a per-row basis during processing.	Adding a new row might add more context to a certain problem.
Should have	FR2.4	The system shall be able to construct several data processing prompt files in case of a variety of information to be obtained.	Sometimes multiple questions could be answered with the same row of data to obtain as much information as possible.
Must have	FR2.5	The system shall be able to test a prompt on a few selected or random rows.	The researcher might want to test a prompt for correctness before trying to run it for hours.
Must have	FR3.1	The system shall include robust output parsing and visualize this in the interface for usability.	Parsing output is necessary for visualization and can make systems more usable.
Should have	FR3.2	The system shall include monitoring of job progress.	Analyses can run for hours, so researchers need to see how far a job has progressed.
Must have	FR3.3	The system shall include a web interface to improve user accessibility and the usability of the system for researchers.	Researchers shouldn't have to interact with code to get results. The interface makes accessing the backend much easier.
Must have	FR3.4	The system shall export the results in a CSV format for R/SPSS/Excel.	Having export options for data is mandatory.
Must have	FR4.1	The system shall include a method to store a template with the derived results for future reproducibility.	Analyses may need to be redone to verify results, investigate outcomes, or reproduce previous research.
Should have	FR4.2	The system shall support pausing, resuming and cancelling an analysis.	Sometimes results might not be usable at all.
Could have	FR4.3	The system shall support multiple LLM endpoints to be shared with different hospitals.	Research is conducted in collaboration with multiple hospitals, so this feature makes collaboration easier.
Should have	FR5.1	The system should include parallelization for faster processing.	Given that multiple tasks need to be performed, parallelizing prompts makes the system fast.

Table 3: Prioritised functional requirements.

Priority	ID	Requirement	Justification
Must have	NFR1.1	The analysis monitoring interface should be easy to understand without requiring technical knowledge.	The primary users are medical researchers with little programming knowledge.
Must have	NFR1.2	The system should include logging of LLM outputs for debugging and reproducibility purposes.	The model might return data that is structured incorrectly, or might receive incomplete data.
Must have	NFR1.3	The system should include proper documentation for future maintainability.	Future developers should be able to clearly understand the code to make changes or improvements.
Must have	NFR1.4	The system shall handle an unexpected LLM output without terminating the analysis.	Robust systems should handle unexpected output without crashing, so that already obtained results are not lost.

Table 4: Prioritised non-functional requirements.

2.4 Deployment and Demonstration

The project was developed locally and deployed to the DZ around week 8, where it was tested with realistic artificial healthcare-oriented data. The deployment used a roughly 500 MB CSV on the hospital machine and worked well in the intended on-premise setting. For a 24B Mistral model on the single RTX 6000 Ada, the relevant expected generation range is approximately 40–60+ tokens per second, depending on quantisation, inference engine, context length, batching, and concurrent requests. This is a hardware-range estimate rather than a universal benchmark. The hospital machine remains isolated, and the documented self-contained deployment procedure supports future maintenance and re-installation by the hospital.

2.5 Risk analysis

The main risks were reviewed during the requirements and sprint planning.

Risk	Likelihood	Impact	Mitigation	Outcome
No approved real patient data for development	High	High	Use realistic artificial/public data; separate technical testing from clinical validation; require DZ validation before use.	Occurred; artificial evaluation used.
Isolated hospital machine prevents remote testing	High	High	Keep the deployment self-contained, prepare reproducible setup instructions, and schedule an on-site trial.	Occurred; on-site deployment testing was used.
GPU memory exhaustion or CPU spillover	Medium	High	Estimate model weights and KV cache, cap workers, keep batch size configurable, and increase concurrency only after monitoring.	Controlled;20 workers at 8k used approximately 42 of 48 GB in the observed stress test, leaving limited headroom.
Hallucinated or malformed LLM output	High	High	Use typed output schemas, Pydantic validation, retries, explicit error states, and human review of results.	Malformed output was retried; semantic correctness remains a user responsibility.
Deployment failure caused by dependencies or configuration	Medium	High	Pin dependencies, provide a `.env.example`, document frontend build and backend startup, and test the complete workflow before handover.	Reduced; the deployment path and limitations are documented.
Scope growth during a short project	Medium	Medium	Use GitLab epics, labels, one-week sprints, MoSCoW prioritisation, and branch-based work.	Controlled through sprint replanning.

Table 5: Project risks, mitigation measures, and observed outcomes.

2.6 Planning and milestones

The project was planned in one-week iterations from requirements analysis to final reporting. Sprints 1–2 established the domain, constraints and MoSCoW requirements; Sprints 3–4 established the architecture and connected the React frontend to the FastAPI backend; Sprints 5–6 implemented output columns, project lifecycle controls, dependencies and large file handling; Sprint 7 focused on deployment, testing and documentation; and Sprint 8 focused on stress testing, report revision and handover preparation. The corresponding detailed weekly summary is in Appendix E. Work was divided by component rather than by isolated ownership: frontend components were developed in parallel with backend services, while the scheduler, LLM integration, tests, report, and demonstrations were shared integration responsibilities. Every sprint ended with a merged, demonstrable state and a review of remaining risks.

2.7 Requirement traceability and status

ID	Requirement	Design component	Evidence/test	Status
FR1.1	CSV retrieval and parsing	Papa Parse, upload validation	CSV/API workflow tests	Met
FR1.2	Normalize categorical values	Enum output validation	Output-schema tests	Partly met
FR1.3	Handle missing values	Per-job missing-value policy	Validation and retry tests	Met
FR2.1	Dynamic prompt construction	Prompt builder	Full project workflow	Met
FR2.2	Include selected row fields	Column selector and prompt builder	Full project workflow	Partly Met
FR2.3	Per-row processing options	Row and filter configuration	Project workflow review	Partly met
FR2.4	Several processing prompt files	Multiple jobs within a project	Full project workflow	Met
FR2.5	Test selected rows	Row-range execution	Project API tests and UI demonstration	Met
FR3.1–FR3.4	Parse, monitor, display and export results	Pydantic schemas, polling, result viewer, CSV export	API and end-to-end tests	Met
FR4.1–FR4.2	Save templates and control lifecycle	Template service and scheduler states	Workflow and scheduler tests	Met
FR4.3	Multiple LLM endpoints	Configurable client abstraction	Configuration review; no second endpoint trial	Partly met
FR5.1	Parallel processing	Async scheduler and workers	Parallel-job and scheduler tests	Met
NFR1.1	Easy-to-understand monitoring interface	React UI and progress view	Acceptance workflow	Met
NFR1.2	Logging of LLM outputs	Metrics and result logging	Ollama metrics and result review	Met
NFR1.3	Documentation for maintainability	README and report	Documentation review	Met
NFR1.4	Handle unexpected LLM output	Pydantic validation and retry	29 backend tests	Met

Table 6: Traceability from requirements to design and verification evidence.

3 General System Design

This section provides an overview of the system layout and the rationale behind the major design decisions. The layout is discussed in a top-down format, beginning with the deployment in the Deventer Hospital infrastructure and ending with the design of the frontend. Everything that is discussed is linked back to the requirements of section 2.3 and to the constraints.

3.1 Architecture Overview

Figure 2 shows how the system is organized from a high level perspective. This section will go through the different technologies and frameworks used, providing a rationale behind the choice, and explain their functionalities. The system consists of 5 logical parts that communicate and run on the hospital machine.

1. *Web frontend:* A browser app used by the researcher. This parses CSV files locally and lets the researcher design project. The researcher can do so by adding jobs, filters and input columns.
2. *API:* A FastApi service [30] that is the only entrypoint of the frontend to the system. The API validates input with Pydantic [29] and handles authentication through the *Auth* module. This then communicates with the database and the scheduler.
3. *Database:* A SQLite database [31] that has schemas to store users, tokens, projects, jobs, prompt templates and results. The database is not allowed to store a CSV with patient data (see 2.1.1).
4. *Scheduler:* this is a component that keeps the runtime state of running projects. This hands batches of work to a number of workers.
5. *LLM:* this is the key component of the system. The project supports Ollama [32], which is serving quantised models, but also other LLMs through API calls.

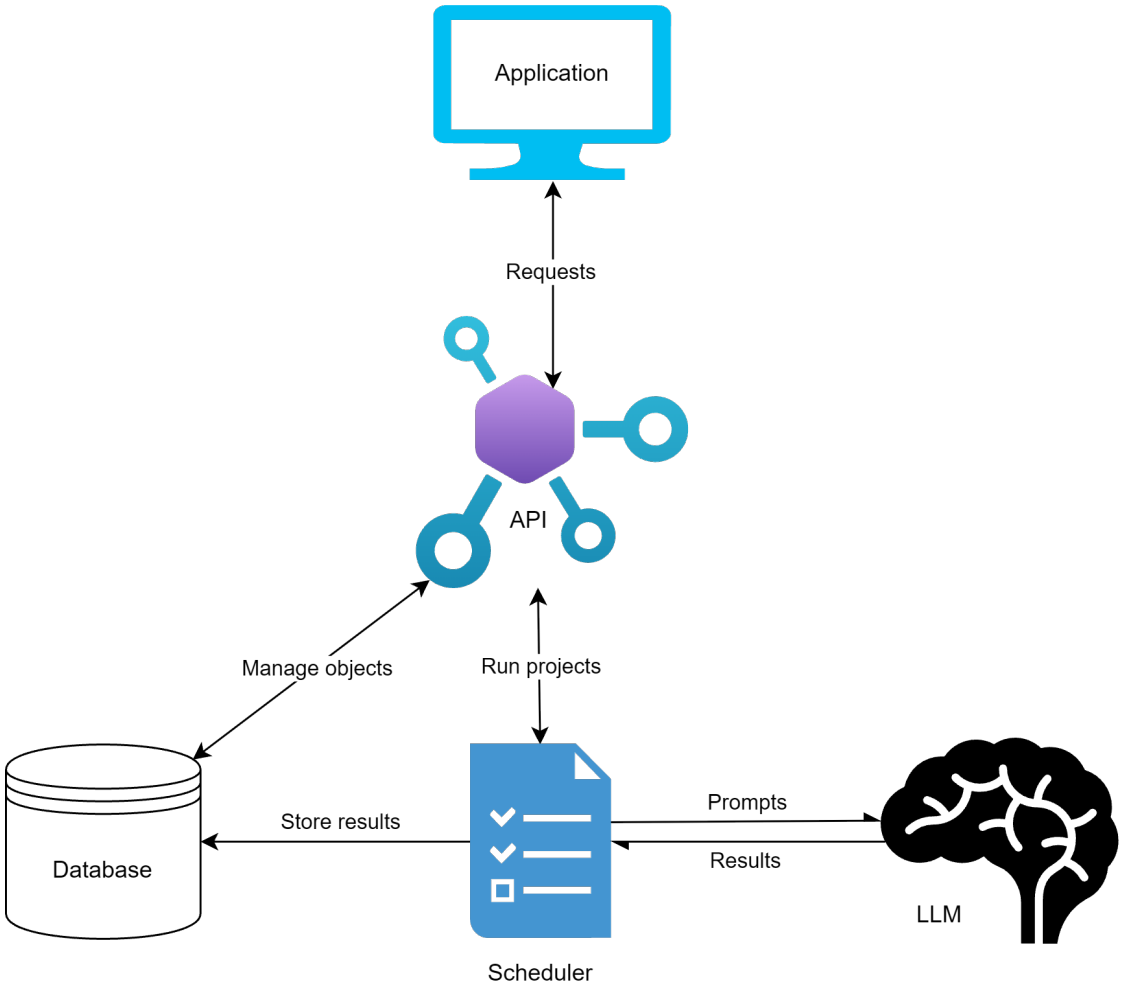


Figure 2: General system overview and data flow.

3.1.1 Deployment view

The hospital machine is isolated from the hospital network and has no remote access (Section 2.1). This means that all components run on the same machine and communicate through the interface.

3.2 Design Decisions

3.2.1 Backend

Fastapi FastAPI was selected for the API because it fits the existing Python service and supports the two backend needs that most directly affect this system: asynchronous request handling and explicit interface schemas. FastAPI also generates OpenAPI documentation and an interactive Swagger UI from the route definitions and Pydantic schemas. This provided a practical way to inspect requests and responses during development and made the frontend decoupled from the backend. Allowing us to work on both parts at the same time.

Rather than implementing our own secure login, the built-in OAuth2-compatible security helpers were used for the bearer-token flow (`OAuth2PasswordBearer` and `OAuth2PasswordRequestForm`); these describe the authentication scheme to the API without requiring an external identity provider. A separate framework or a handwritten API layer could have provided similar endpoints, but would

either introduce another technology into the existing Python deployment or require duplicating schema, documentation, and security.

Pydantic Pydantic was chosen as the boundary between untrusted CSV/API input and the application. Its typed models make required fields, enumerations, and malformed values explicit, so invalid requests are rejected before reaching the scheduler or database. Other than input schemas, the backend also uses Pydantic for general type checking and output validation for the LLM. This is especially important because LLM output is not guaranteed to follow the requested schema.

Sqlite For the database we chose SQLite because it is hosted by the application environment itself: it requires no separate database server, network connection, or operational service on the isolated hospital machine. This makes it much easier to setup as you do not need to setup the database server. Other options like PostgreSQL might be better if the database received more traffic, but our system will be used by a small group.

3.2.2 Frontend

The frontend uses React with Vite and TypeScript. React was selected because the team already had experience with component-based interfaces, reducing implementation risk and making tables, dialogs, status cards, and form controls reusable. Vite provides fast startup and hot-module replacement, which shortened the feedback loop during UI iterations. TypeScript makes API responses and component properties explicit, ensuring errors from backend schemas are immediately found.

Shadcn UI and Tailwind CSS were used to keep the interface consistent while still allowing the team to adapt layouts to the researchers' workflow. Papa Parse reads CSV files incrementally, avoiding the browser-memory problem encountered with session storage and preventing the page from freezing on large files. React Router keeps navigation predictable, TanStack Table provides sortable and paginated data tables, and Lucide supplies familiar icons with consistent sizing. These tools were chosen for usability as well as development speed: the interface presents complex job state and column selection without exposing implementation details to the researcher.

3.3 Backend design

3.3.1 Domain model

The DB schema (Figure 3) follows a structure that mirrors how the researchers at the DZ think about their work:

Project A collection of questions applied to a dataset. A project contains one or more jobs.

Job One task, consisting of a prompt, CSV, and a typed output definition (for example boolean, number, category or text). A job may declare other jobs as *sources*, meaning that it uses their output as input.

*Template (*sjabloon*)* A stored, reusable prompt definition without patient data (FR4.1). Templates are copied into a project job when used; the template catalogue therefore does not need a direct job foreign key.

Results The CSV output per row and job, keyed only by the identifier column chosen by the researcher.

A job can only start when all its sources are complete. This gives sequential prompting (reusing the output of one prompt in another) without a separate concept.

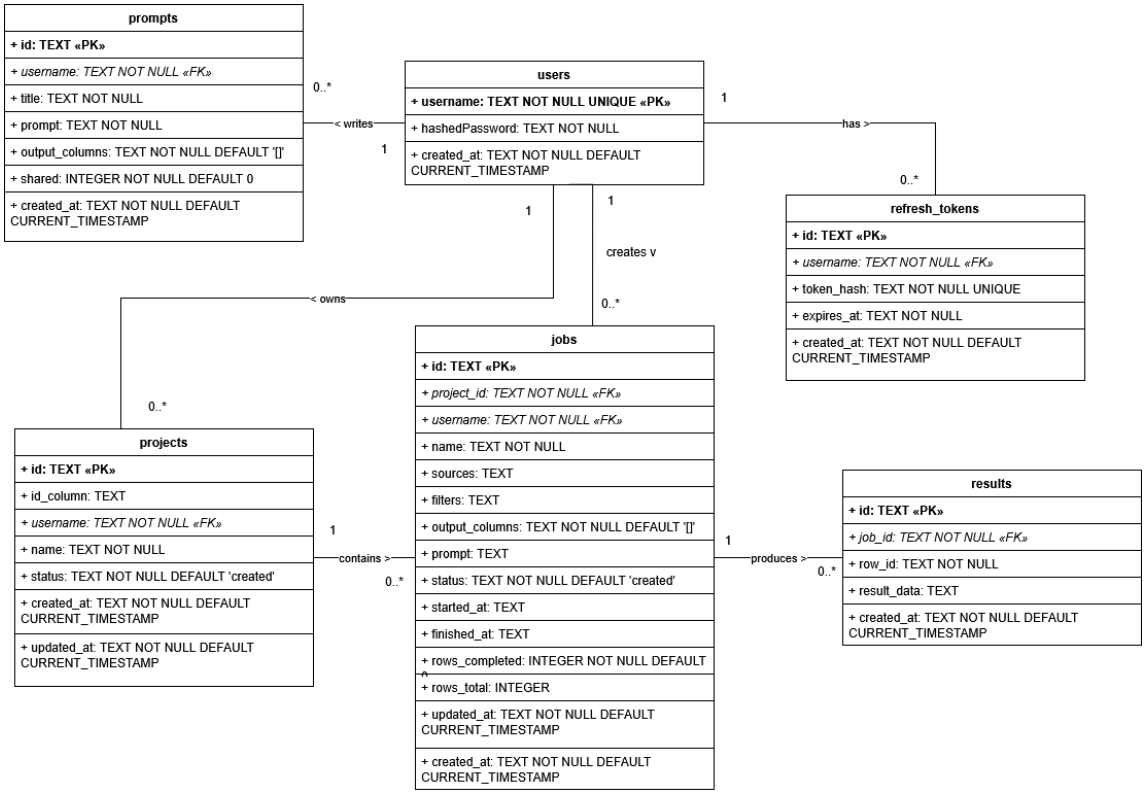


Figure 3: Domain and persistence classes used by the backend.

3.3.2 Authentication

The DZ single sign-on cannot be used, so the platform has its own profiles. The login flow (Figure 4) uses a short-lived access token and a longer-lived refresh token according to the principles of JWT [39]. The refresh token is stored in the database, so that it can be checked and revoked on the server; the access token lets the API authorize requests.

Passwords are stored only as salted hashes using the recommended Argon2id algorithm from `pwdlib`. Every project, job and result belongs to a single user. Every endpoint except registration, login and token refresh requires an authenticated user, and is limited to that user's own projects, jobs and results. This allows several users to work on one machine without seeing each other's work simultaneously.

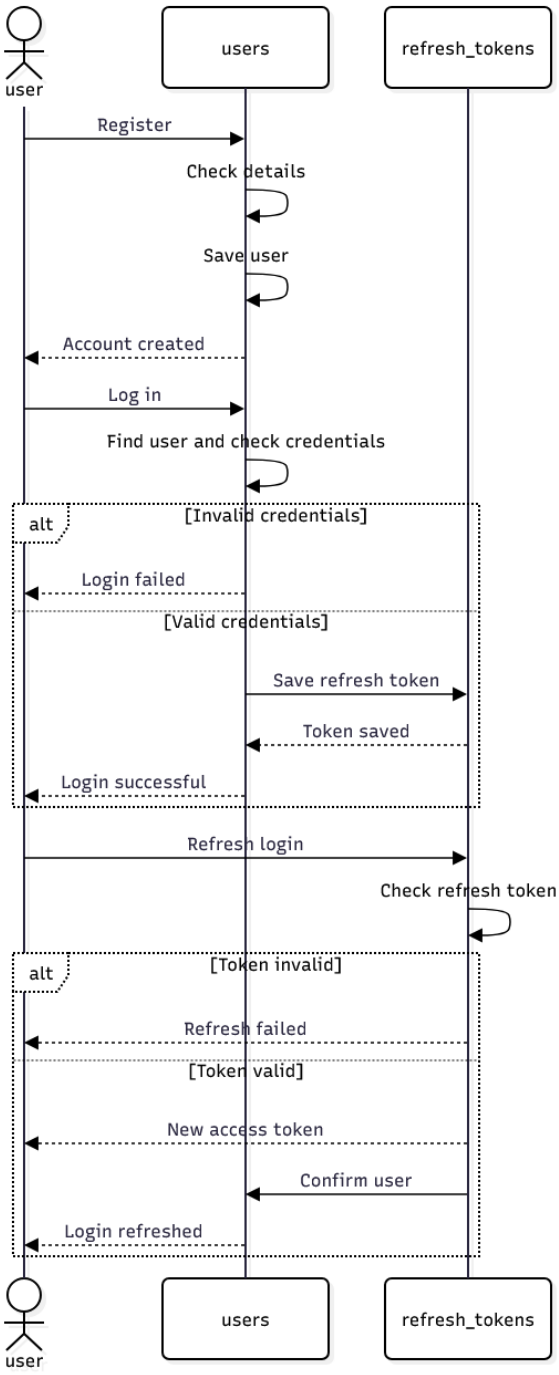


Figure 4: Sequence of registration, login, and token refresh.

3.3.3 API

The API is split into four components: authentication, projects and execution, jobs and templates. Each request in the API expects an access token to verify the identity of the user.

Authentication. In here researchers register with a username and password. Then they log in to receive a short-lived access token. A longer-lived refresh token is set as a cookie, and the frontend uses it to obtain a new access token without asking the researcher to log in again. This resembles the flow described in Section 3.3.2.

Projects and execution. These cover everything a researcher needs for a project. Projects can

be created, listed, renamed and deleted. Their jobs and results can also be retrieved. A researcher can also run, pause, resume and cancel operations, as explained in table 7. A status endpoint returns the progress of projects and their jobs, as defined in Section 3.3.5, and is polled by the interface.

Jobs. Jobs (3.3) can be created, read, updated and deleted. The results per job can also be obtained.

Saved prompts. Saved prompts are the reusable templates (*sjablonen*) of requirement FR4.1. They can be created, listed, updated and deleted.

Note that every operation is scoped to the authenticated user, so researchers sharing one machine cannot see or modify each other's projects, jobs, etc. The run and resume endpoints validate the project and the CSV (Section 3.3) and hand the work to the scheduler. The frontend then polls the status endpoint for progress, so the interface stays responsive during analyses that run for hours (FR3.2). Also, errors are returned immediately with a clear message through the built-in error handling. The development `/debug` router is excluded from the hospital handover configuration because its runtime view could expose source text.

3.3.4 Run-time validation of input

When a project is run, the API reads the CSV contents and loads the project and its jobs from the database, then checks the following before queuing:

1. The project and jobs are runnable, eg. every job has a prompt and output definition, dependencies reference existing jobs, the input and output columns are not cyclic, and
2. the uploaded data is valid: it contains at least one row, the project's identifier (ID) column exists, and every identifier is non-empty and unique. Missing values in other columns are allowed.

The ID checks are essential because results are matched back to rows only through that ID. A failed check returns an error to the researcher immediately. (FR1.1, FR1.3).

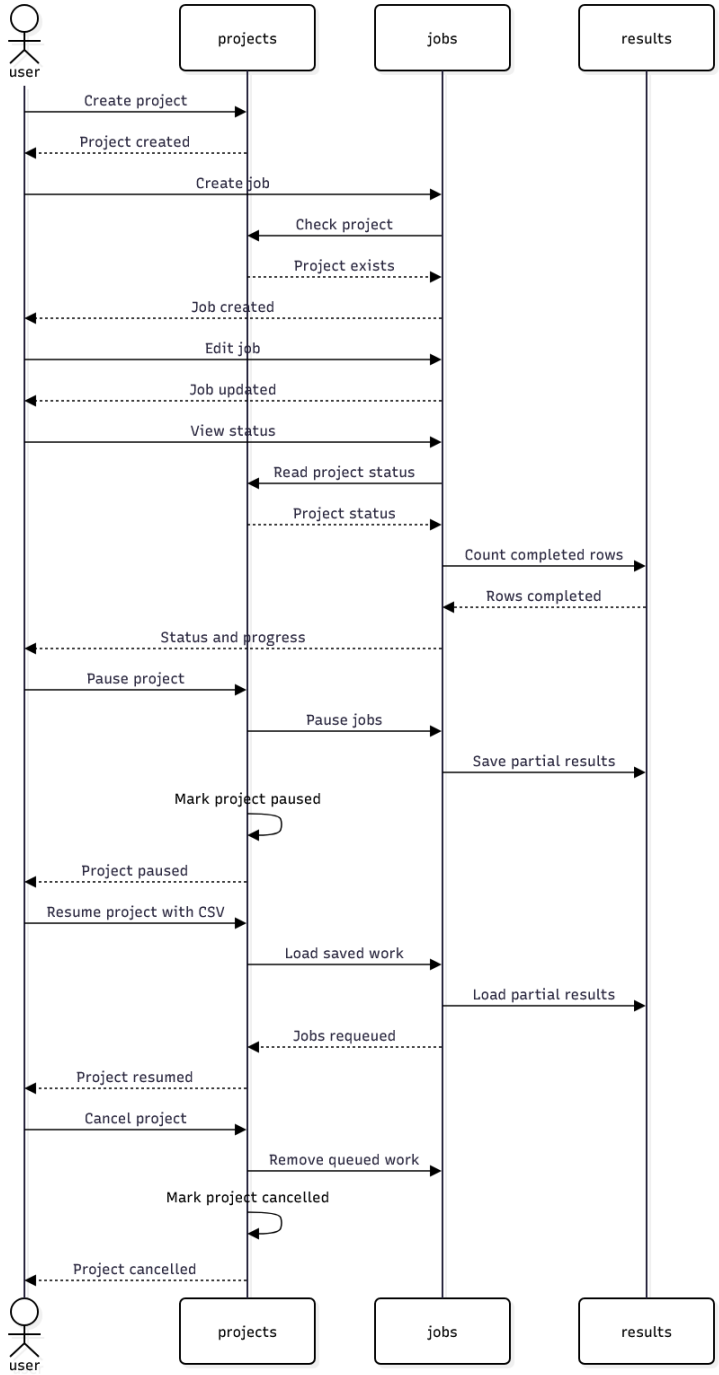


Figure 5: Project and job management sequence using the backend objects.

3.3.5 Scheduler

States A central architectural idea is the separation between persistent state and runtime state.

- The persistent state lives in the database: definitions (projects, jobs, prompt templates), users and tokens, project status and finished job results. This are stored, even with a restart of the server.
- The runtime state lives in the scheduler's memory: these are the waiting, active, and in flight batches. It exists only while a project is running, and if completed are removed from runtime.

This split is what makes the privacy and ethical constraint enforceable by default: the only component that ever holds patient text is the runtime state, and that state is volatile. Moreover, only processed data is stored, so this does not violate any issues raised beforehand.

State Management A project is always in exactly one state. Table 7 lists the states and the events that move a project between them.

State	Meaning	Leaves the state by
Queued	Runtime project exists; jobs wait because their sources are not ready.	Sources become ready → Running; cancel → Cancelled.
Running	At least one job is in the active queue or being processed.	All jobs complete → Completed; a job exhausts its retries → Failed; pause → Paused; cancel → Cancelled.
Paused	Partial results saved, runtime state removed.	Resume (with CSV) → Queued/Running; cancel → Cancelled.
Completed	Merged results saved, runtime state removed.	Terminal (a new run is possible).
Failed	A job failed after the retry limit.	Terminal; results of finished jobs are kept.
Cancelled	Queued work removed, partial results stored.	Terminal.

Table 7: Project lifecycle states.

Queues The scheduler has an active and a waiting queue. The waiting queue contains all jobs for which its' parent (sources) jobs have not been completed yet. Once all sources have been completed, a job is moved to the active queue. While in the active queue, the workers can ask the scheduler for a new batch (a row of data). The scheduler then picks a job from the active queue (by the round robin principle) and gives the batch to the worker. This was chosen due to it being a simple implementation and it fairly spreads the LLM to all jobs. Once a job completes, it is removed from the active queue.

Batching and the Worker Loop The scheduler splits the rows of every active job into batches of the configured batch size (see 3.3.5). A fixed number of asynchronous workers, set per model (Section 3.3.7), run concurrently in a single asyncio event loop [25]. Each worker repeats the same loop (Figure 6):

1. **Ask** the scheduler for a work batch, which contains the rows of data, prompt and output schema.
2. **Build** the prompt for each row from the job's template, its CSV columns and the outputs of its source jobs (Section 3.3.7).
3. **Call** the model through the LLM client and wait for the response. While a worker waits, the event loop serves the other workers, so the GPU stays busy.
4. **Validate** the response against the job's typed output definition. Valid values are stored in the runtime project and `rows_completed` is updated, which drives the progress calculation (Section 3.3.5).
5. **Retry** the batch if the response is invalid or the request fails, up to a configured retry limit. If the limit is exhausted the job, and therefore the project, becomes *Failed* (Table 7). Results of rows and jobs that were already finished are kept.

An invalid model response never terminates the analysis. It only causes a retry of the affected batch, which satisfies NFR1.4. Because completed batches are recorded per row, a pause or resume only has to process the missing rows again (Section 3.3.5). When the last job of a project completes, the results are saved to the database, but the CSV is removed from runtime.

Batch size The batch size is a variable in a Job that states how many rows should be processed at once in the same LLM request. A larger batch reduces scheduling overhead but loses more work when a request fails. A smaller batch gives smoother progress feedback and limits the cost of retries. The batch size is therefore a configuration value to tune with the target model and hardware, because the application has to segment rows into batches that fit in the context window.

Progress calculation Progress is defined so that it can be computed in the same way from saved and from live data:

$$p_{\text{job}} = \begin{cases} 1 & \text{if the job is complete,} \\ \dfrac{\text{rows_completed}}{\text{rows_total}} & \text{otherwise,} \end{cases} \qquad p_{\text{project}} = \frac{1}{n} \sum_{i=1}^{n} p_{\text{job},i} \times 100$$

The status operation first checks the status from the database. If a runtime project exists, the live scheduler status overwrites it, otherwise the saved status is returned as is. The average over jobs weights all jobs equally, regardless of how many rows they have or how long they take.

Lifecycle operations The scheduler has to deal with pausing, resuming and canceling certain projects. Or even worse: a project might have a single job that fails and has to be rescheduled.

Pause. The scheduler saves partial results to the database, marks the project paused and removes it from both queues, which also frees the runtime copy of the CSV. Because partial results contain only outputs and identifiers, no source data is persisted.

Resume. The researcher uploads the CSV again (the platform never kept it). The API loads the saved work, recreates the runtime project and the scheduler restores completed rows and dependencies, so only the missing rows are processed again.

Cancel. The scheduler removes queued work, the API marks the project cancelled and results of jobs that had already completed are retained.

Requiring the CSV to be uploaded again on resume is the visible cost of the no-source-data constraint. The identifier check (Section 3.3) is repeated on resume, so a file with different identifiers cannot silently be mixed with saved results. A future version should additionally persist a non-reversible file fingerprint to detect changes without storing source text.

3.3.6 Scheduler and worker design

The scheduler replaces the manual worker loop of the existing script (limitations 2 and 3 in Section 1.6). Figure 6 gives the sequence; the control flow is summarised below. The sequence uses backend objects that correspond to the domain and persistence classes rather than treating the data department as an actor. The data department supplies data before the use case starts; it does not trigger the runtime interaction shown here.

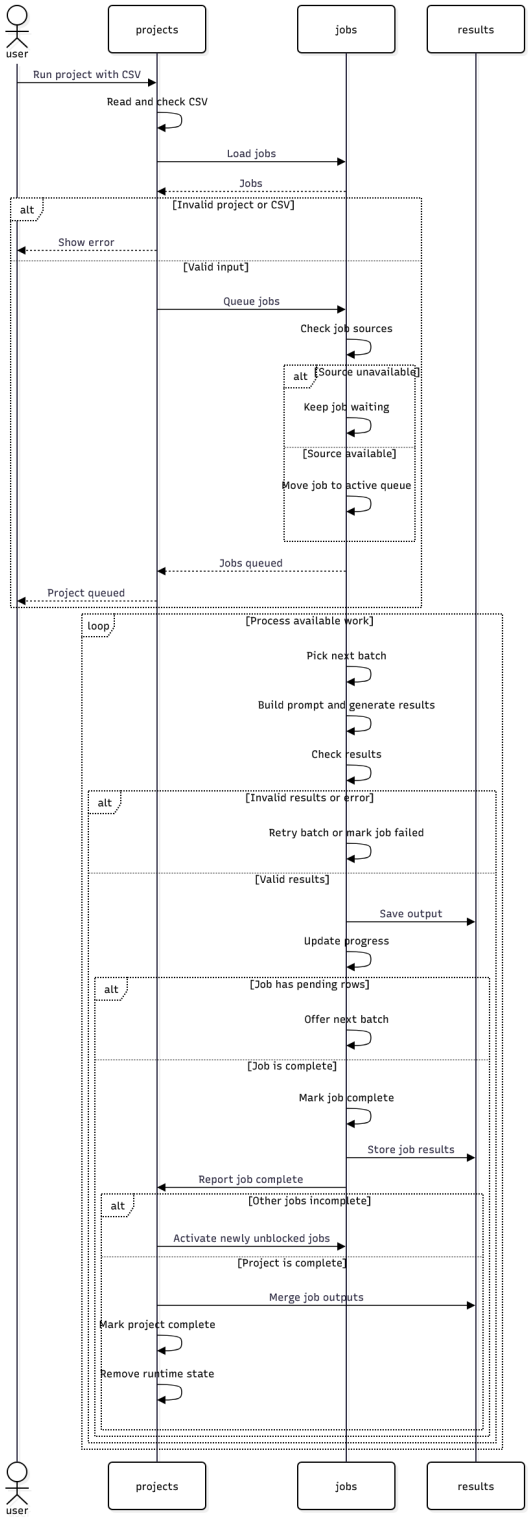


Figure 6: Scheduler and worker interaction derived from the backend classes.

3.3.7 LLM Layer

Model abstraction The existing API only works with one hardcoded model and endpoint. The new design puts a thin *LLM client* interface between workers and the model server. A model configuration contains an endpoint URL, a model name, the context length and generation parameters (temperature, maximum output tokens). Because Ollama exposes models through a uniform API

[32], changing the model is a configuration change, not a code change (FR4.3). The interface is intentionally narrow ("given a prompt, return text and token statistics"), so other servers with a compatible API could be added later.

System Prompt A prompt was supplied to the system in the backend. This is concatenated to a system prompt (Appendix C). This is important as it ensures the LLM's responses are within a specific context. In a study conducted by Kong et al [15], agents that were assigned roles, performed better on multiple benchmarks compared to standard zero shot approaches.

Prompt construction When the worker receives the rows and prompt, it creates the prompt that will be sent to the LLM. For each row the worker substitutes the placeholders in the prompt (FR2.1, FR2.2), cleans the text (whitespace and control characters) and appends an output instruction that states the expected type and allowed values. Rows with missing values in a referenced column are handled according to a defined policy (skip, use an explicit "unknown" marker, or fail the row), which is chosen per job (FR1.3).

Typed output and validation The model is instructed to answer in a fixed machine-readable format, and the answer is validated against the job's output definition with Pydantic. A value is accepted only if it has the right type and, for categories, lies in the allowed set. Anything else counts as an invalid result and is retried. Validation guarantees *format*, not *correctness*: a well-formed but wrong answer passes. This distinction is deliberate and is why the requirements treat full validation of results as a recommendation (Section 2.1) and why Chapter 5 measures accuracy separately.

GPU memory budget GPU memory has to hold the weights and the KV cache of every request that is processed in parallel (Table 2, Section 1.3.4). This relationship also applies when the model is served through Ollama: Ollama maintains a full context slot for each parallel request. The context length can be set with `OLLAMA_CONTEXT_LENGTH` or the API's `num_ctx` option, while the number of simultaneous requests is controlled by `OLLAMA_NUM_PARALLEL`. Increasing either value increases memory use; a 32k context with four parallel requests therefore allocates four 32k slots, not one 32k slot divided over four workers. The same rule means that 20 workers at 8k allocate 20 separate 8k slots. The KV cache needs, per token [40],

$$m_{\text{token}} = 2 \cdot L \cdot H_{kv} \cdot d_h \cdot b,$$

where L is the number of layers, H_{kv} the number of key-value heads, d_h the head dimension and b the number of bytes per value. For a 16-bit cache ($b = 2$), Mistral Small 3 (24B), with $L = 40$, $H_{kv} = 8$ and $d_h = 128$, needs approximately 160 KB per token. The theoretical payload for one 8,192-token sequence is approximately 1.3 GB; in the deployed Ollama configuration, four full context slots require approximately 5 GB of KV cache. The following deployment estimates combine the approximately 14 GB Q4_K_M weights with the relevant number of parallel KV-cache slots and approximately 1–3 GB of runtime overhead.

Context	Workers	KV cache	Total VRAM	Interpretation
4k	4	≈2.5 GB	≈18–19 GB	Low memory use; suitable for short prompts.
8k	4	≈5 GB	≈21–22 GB	Conservative operating point for normal processing.
8k	20	≈27 GB	≈42 GB	Observed stress-test result on the RTX 6000 Ada; only a small reserve remained.
16k	4	≈10 GB	≈26–27 GB	More context headroom while retaining substantial GPU reserve.
32k	4	≈21 GB	≈37–38 GB	Still fits on a 48 GB card, but leaves less room for runtime variation.

Table 8: VRAM and KV use per amount of workers for `mistral-small:24b`.

Worker recommendation The theoretical cache calculation is an upper bound, not a worker setting. For a model with GPU memory budget M, model weights W, per-sequence KV cache K, and a safety reserve R for the operating system, Ollama runtime, temporary buffers, and allocation variation, the memory-based upper bound is

$$N_{\text{memory}} = \left\lfloor \frac{M - W - R}{K} \right\rfloor .$$

Here, N_{ollama} is the configured maximum number of concurrent requests for the model; the application worker count should not exceed it. For this project, $M = 48$ GiB and $R = 8$ GiB is a deliberately conservative starting reserve. Using the 14 GB weight estimate and the approximately 1.25 GiB 8k per-worker cache estimate, the theoretical arithmetic gives

$$\left\lfloor \frac{48 - 14 - 8}{1.25} \right\rfloor = 20.$$

This calculation is only a rough upper bound and does not account for all runtime buffers. A lower worker count remains a safe default when workload variation or multiple users are expected. The actual allocation should be checked with `ollama ps` and `nvidia-smi` after loading the model. If lower memory use is required, flash attention and an 8-bit KV cache can reduce the 32k four-worker estimate to approximately 30 GB, subject to validation of quality and latency.

3.4 Frontend Design

3.4.1 Frontend Architecture

In the React frontend application, a consistent structure is applied. This was done to make sure the system is maintainable and easy to extend. The source folder contains subfolders for each of the main pages of the application. Each main page folder consists of one TSX file that contains the main layout of the page, and some also contain a components folder consisting of smaller TSX files. These components are used to keep the code organised and improve its readability and maintainability. Aside from to that, at the same level as 'src', there is a general components folder containing components that are reusable across all pages. That is the location where Shadcn

places a component when it is installed. All of this can be checked below where a markdown layout is written down.

```
frontend/
 components/
 src/
     pages/
        homepage/
           HomePage.tsx
           components/
        helppage/
           HelpPage.tsx
        projectpage/
           ProjectPage.tsx
           components/
        templatepage/
           TemplatePage.tsx
        loginpage/
           LoginPage.tsx
     app.tsx
```

3.4.2 Task page

At first, a general overview of outstanding jobs was thought of. This should have a distinction between different projects, as well as providing some insight in the progress of the task, see section 4.2.3. This page is important as it allows researchers to see a general overview of all their active jobs and their respective statuses. This way there is no need to search in your projects to find the job you are interested in.

Initially, the task overview page was designed to have cards actively displaying statistics such as total personal active prompts, total prompts running etc. (see figure 7). However this was removed as the info served no real purpose but would add code. The rest of the initial design is clearly similar to the final page. A table containing a researcher's personal prompts were instead displayed. To keep it simple, there are no action buttons in this tasks table, because it might not be known well enough to the researcher what exactly the job - with possible relations to other jobs - is (doing), that he is performing an action on.

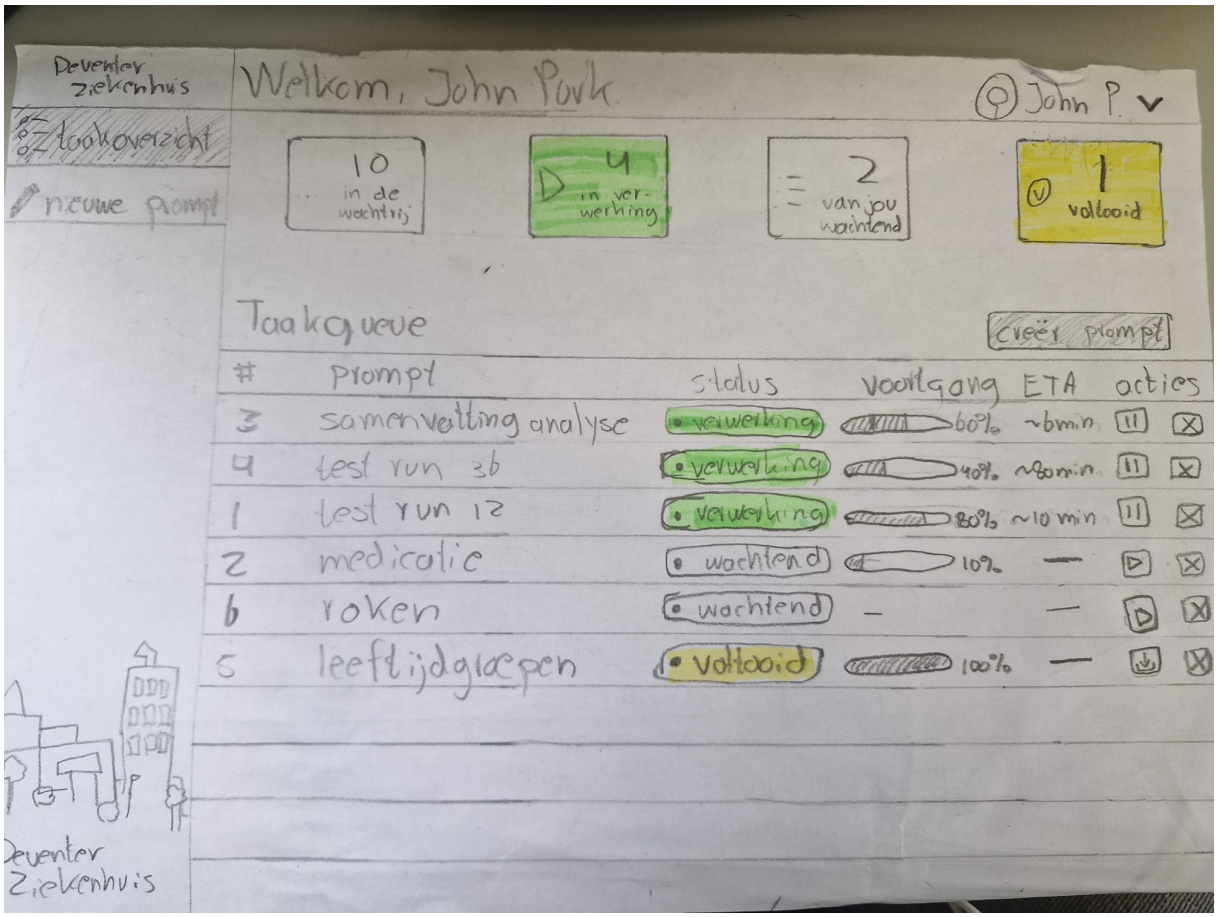


Figure 7: Original sketch of task overview

3.4.3 Project Page

Secondly, an overview of all the projects a researcher has was needed. This page shows a table with the logged-in user's projects together with several actions. The table uses few columns: project title, amount of finished tasks, project status/progress, last edited date, and an actions column. All supported project actions are placed in the table, allowing users to manage projects at a high level. To support researchers with many projects, a search bar performs a text search on project titles.

3.4.4 Job Page

In order to perform investigations on the CSV file, a proper prompt creation page was made. This page is the core of the application: it allows a researcher to add a prompt, choose the rows on which it operates, select input columns, define typed output columns, and reuse outputs from earlier prompts. The first design used a simple prompt title and question popup; it evolved into the current project-based flow with filters, dependencies, previews, saved templates, and result viewing. The researcher selects the model configuration and temperature in the job settings; the configured value is passed to the LLM client for every batch. A lower temperature is appropriate for deterministic extraction, while a higher value is available for exploratory prompts.

3.4.5 Prompt Templates Page

Finally, to allow researchers to easily edit and save prompts for future use, a prompt templates page was created. Originally, this page was designed to also test singular prompts, see figure 8. However this was not necessary as the job page also implements this. This page shows an overview of prompts created by the user, or prompts shared by other users (researchers). Aside from that, they can create, edit and delete prompts as needed.

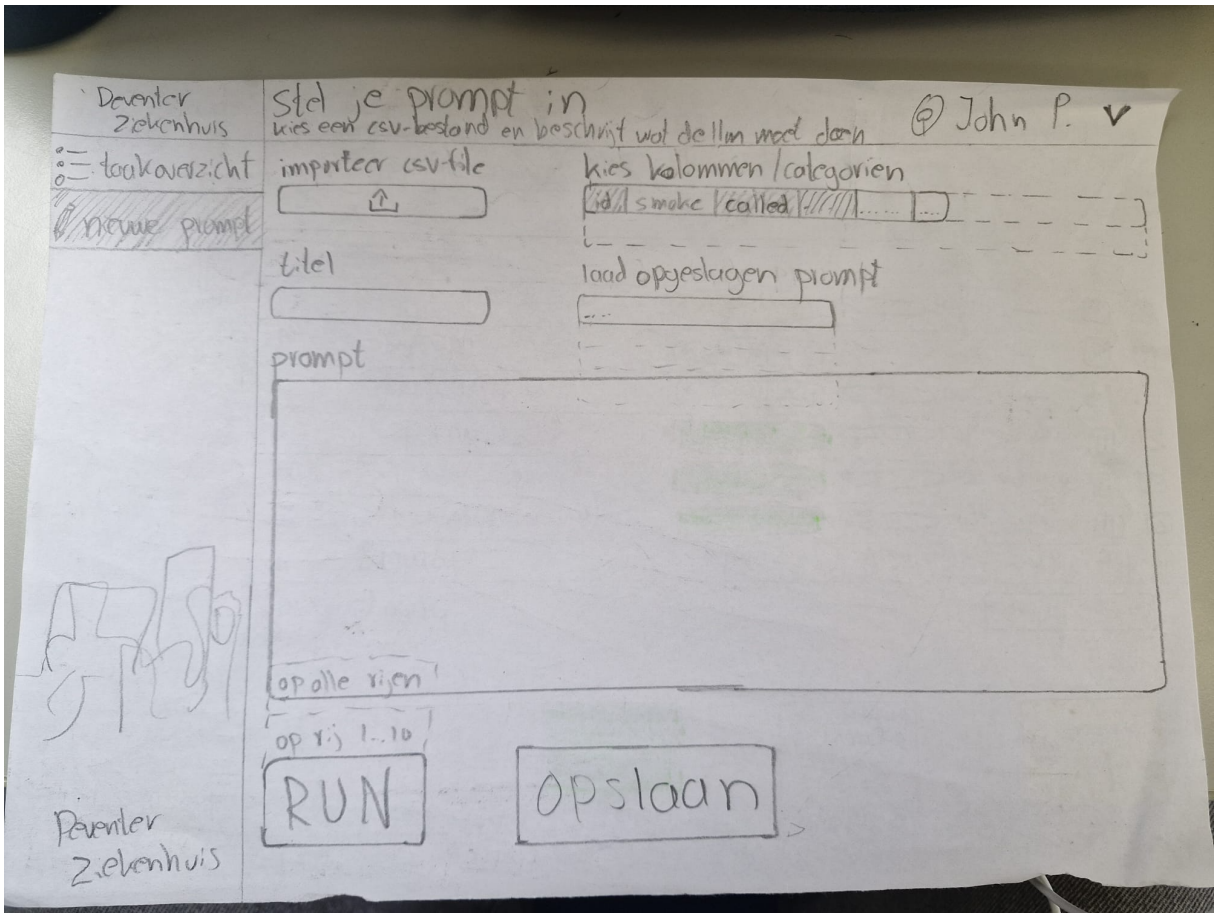


Figure 8: Original new prompt page sketch

4 Detailed Implementation

4.1 Backend

The backend is implemented as a Python 3.13 FastAPI application. It separates HTTP routers, service functions, Pydantic request schemas, domain models, SQLite helpers, authentication, and LLM runtime components. This separation was tied to two project constraints: six students had to work in parallel during one-week sprints, and the hospital needed a self-contained service that could run without a cloud identity provider. A single FastAPI module would have coupled request parsing, ownership checks, database operations, and long-running model calls; that alternative would have made parallel changes and scheduler tests more fragile. The trade-off is more explicit conversion between database and runtime models. The structure is visible in `app/projects/router.py`, `app/jobs/router.py`, `app/llm/scheduler.py`, and `app/db/database.py`. The resulting module boundaries made the code easier to navigate and allowed the team to change the API, scheduler, and database independently.

4.1.1 Authentication and authorization

Registration works with lower-cases usernames, requires at least eight password characters, and stores the result of `PasswordHash.recommended()` from `pwdlib[argon2]` (version 0.3.1 in `requirements.txt`). The current report identifies this deployment as Argon2id; its work factor is library-managed rather than hard-coded. A password hash was chosen instead of encryption because the service only needs to verify a password and must not be able to recover it. Login issues a JWT access token with a default lifetime of 15 minutes and sets a refresh token in an HTTP-only, `SameSite=Strict` cookie with a default lifetime of 24 hours. The shorter bearer-token lifetime limits exposure, while the cookie keeps the refresh token out of frontend JavaScript. Refresh tokens are rotated during refresh and only a SHA-256 hash of each current refresh token is stored in SQLite. So in conclusion the implementation combines short-lived bearer credentials with a server-controlled refresh credential rather than placing both tokens in browser storage.

4.1.2 Database and service boundary

The database layer uses Python's built-in `sqlite3` module and parameterised SQL instead of an ORM. SQLite was retained because the hospital prototype already used it, the deployment is on one local machine, and PostgreSQL would have introduced another service to install and maintain on an isolated machine. The trade-off is that schema evolution uses `CREATE TABLE IF NOT EXISTS` and targeted `ALTER TABLE` checks rather than a migration framework. Tables store users, refresh tokens, projects, jobs, prompt templates, and processed results. The uploaded CSV is read into Pandas for a run and remains in scheduler memory; it is not inserted into the database. Ownership is enforced by passing the authenticated username into service and database queries, rather than filtering only in the frontend. The evidence is `app/db/database.py`, especially `init_db`, `get_conn`, and `runQuery`, together with `test/test_database_full.py`.

4.1.3 LLM client and prompt construction

In the hospital deployment Ollama is used. The OpenAI-compatible client path was used during development to test the modular endpoint design without changing scheduling code. The `build_prompt` function receives the selected input rows, identifier column, and declared output columns, then adds a JSON schema and instructions to return one object per input row. This directly supports typed, exportable results (FR3.1 and FR3.4), validatable and exportable to CSV. Provider format controls are an additional aid, not a substitute for validation. The Ollama path sends `format=ITEMS_SCHEMA`, the thinking setting, and the configured temperature; the OpenAI-

compatible path sends `response_format=json_object`. Both paths use the configured model and endpoint so the scheduler does not depend on one provider. The alternative of embedding provider-specific HTTP calls in the scheduler was rejected because it would make endpoint changes affect queue logic. The current worker selects the foreign endpoint only when `enable_foreign_llm` is true; the hospital configuration keeps that setting false and therefore uses Ollama. The evidence is `call_ollama`, `call_llm`, and `build_prompt` in `backend/app/llm/worker.py`.

4.1.4 Boundary validation and typed outputs

Pydantic validates requests before they enter the runtime. Discriminated unions distinguish project-column sources from previous-job sources and distinguish `in` from `not_in` filters. `OutputColumn` allows only string, integer, number, boolean, or enum outputs, rejects invalid enum definitions, and makes nullability explicit. Before queueing, `validate_output_columns` rejects empty or duplicate output names. After a model response, `Scheduler.complete_work` checks every row identifier exactly once, rejects unknown identifiers and columns, requires non-nullable values, and calls `OutputColumn.check` for primitive and enum types. This second check is necessary because valid JSON or HTTP 200 does not prove that the model answered the requested rows. Failed work is requeued and fails after the configured retry count is exceeded. The Filters and configured categorical mappings provide the normalisation required by FR1.2; the hospital must configure the accepted representations for its own data sources. Additional per-row context and options are supplied through the prompt configuration and template controls, supporting the intended FR2.4 workflow. The trade-off is that one invalid response causes the whole claimed batch to be retried, but this avoids silently exporting malformed research data. The implementation is in `app/models/job_output.py`, `app/jobs/schemas.py`, and the scheduler's `complete_work` and `fail_work`.

4.1.5 Scheduler, batching and concurrency

The scheduler keeps separate in-memory waiting and active deques. A job becomes active only when project columns and all earlier job outputs are available in case of a job dependency. Each worker is an asyncio task created at application startup; it obtains work, awaits the model request, and returns the result. Active jobs are rotated in round-robin order so one large job cannot continuously take the first position. Runtime identifiers on projects and work batches prevent a worker from applying stale work after a resume. A batch is the rows included in one LLM request: `get_work` claims one pending row and up to `job.batch_size-1` additional rows. The current `backend/.env.example` sets `BATCH_SIZE=50`, and the setting is configurable without editing the worker. It also sets four worker tasks, matching the four-request hardware policy, although that limit is not enforced by code. Larger batches reduce HTTP overhead but increase prompt and KV-cache usage; smaller batches reduce failure granularity but increase request overhead. The retry limit is configuration-driven through `LLM_RETRY_ATTEMPTS`; the current example sets it to three. These are testing values selected for the University of Twente UTM2 model, not claims about the hospital's final production configuration. The LLM timeout was increased after long requests left projects stuck. The 240 second timeout is also a safe upper bound for development with an online compatible endpoint when a provider responds slowly or applies rate limiting. The hospital's Ollama path does not depend on that external rate-limit behaviour.

4.1.6 Pause, resume, cancel, progress and logging

Pause persists completed results and removes temporary runtime state. Resume requires the CSV to be uploaded again because source data must not be persisted. Unfinished rows are reconstructed while completed outputs are retained. Cancel removes runtime work while retaining database records and completed results. This is less convenient than storing the source file, but it

follows the privacy constraint directly. Runtime identifiers prevent stale work being applied after resume.

Frontend, the progress is fetched by calling `/api/project/status` yielding `rows_completed` and `rows_total`. The project table polls every second, while the results viewer polls every five seconds for running or queued jobs. Polling was simpler than websockets for a single isolated machine and coarse row progress, although it creates repeated requests and display latency. The worker currently prints worker, job, row-count, duration, and exception information in the terminal. The `/debug` router is included for development and testing. Its endpoints serialize runtime dataframes and can therefore expose patient text if reachable. Hence it is excluded from the hospital deployment configuration, although it remains available for local testing.

4.1.7 Deployment

The documented deployment creates a Python environment from `requirements.txt`, builds the React/Vite frontend into `frontend/dist`, creates `backend/.env` from the example configuration, and runs `backend/main.py`. FastAPI then serves both the API and the built frontend. This native path avoids a second database service and matches the hospital's local Ollama arrangement on the isolated machine. Docker was discussed but was never used. No Dockerfile or Docker deployment path exists in the repository. The trade-off is that upgrades and process supervision remain installation tasks for the hospital. Evidence is in `README.md`, `requirements.txt`, `backend/.env.example`, and the static mounting code in `backend/main.py`. The final frontend after deployment will be discussed in the next section.

4.2 Frontend

This subsection guides through the frontend application, including its pages, dashboards, dialogs, and reusable components. The main pages are the login/registration page, task overview, projects overview, jobs page, prompt templates page, and help page.

4.2.1 Frontend implementation choices

The frontend uses React with Vite as its build tool, TypeScript, React Router, and TanStack Table. React was chosen because the interface contains stateful projects, jobs, prompts, and result views rather than a sequence of independent page. Component state can update one job table without reloading the whole application. Vite is a fast and lightweight build tool supporting Hot Module Reload which makes development convenient. It produces a static build that `backend/main.py` can serve from the same origin, which avoids requiring a separate frontend server on the isolated hospital machine. The trade-off is that the build must be regenerated after frontend changes.

CSV handling was changed in response to large file behaviour. The first implementation stored CSV content in browser `sessionStorage`, but large files exceeded browser storage limits. Papa Parse now reads files in chunks, validates headers, and creates a smaller row-range file when the user selects a subset. A web worker was tested to keep the page responsive, but the production build could not use that mode reliably. The current code therefore keeps `worker: false` while using 2 MiB chunks (`frontend/lib/csv.ts`). This is a deliberate compatibility trade-off, not evidence that parsing is offloaded from the main thread completely. A production-compatible worker bundling approach should be revisited before very large files become a routine workload.

The interface polls job status instead of maintaining a live socket because the backend already exposes a compact status endpoint and the deployment is single-machine. Filters and job dependencies are represented as typed controls: a job can select the projects' CSV columns or outputs from an earlier job, and filters are sent using the same discriminated shape as the backend. Templates were retained as shared reusable configurations, while the separate test-prompt action

was removed as this flow required too drastic changes to the current flow. Row-range selection is now part of running a project. Dutch labels and error messages were kept because the intended users are researchers at a Dutch hospital. For example, CSV validation errors are presented in `frontend/lib/csv.ts`. The frontend evidence is `frontend/src/pages/projectpage/components/JobsView.tsx`, `frontend/components/NewPromptPopup.tsx`, and `frontend/lib/api.ts`.

4.2.2 Login/registration page

The very first page the user will land on is the login and registration page. Users can sign up here with just a name and a password which should be of length greater than 8 characters. Within the scope of this system no email verification or strict password constraints were required. As shown in Figure 9 below, the user just has to repeat their password once in order to make sure they will remember it and did not make any typos. After hitting 'Registreren' they are logged in with their entered credentials. This brings them to the first dashboard page: The task overview.

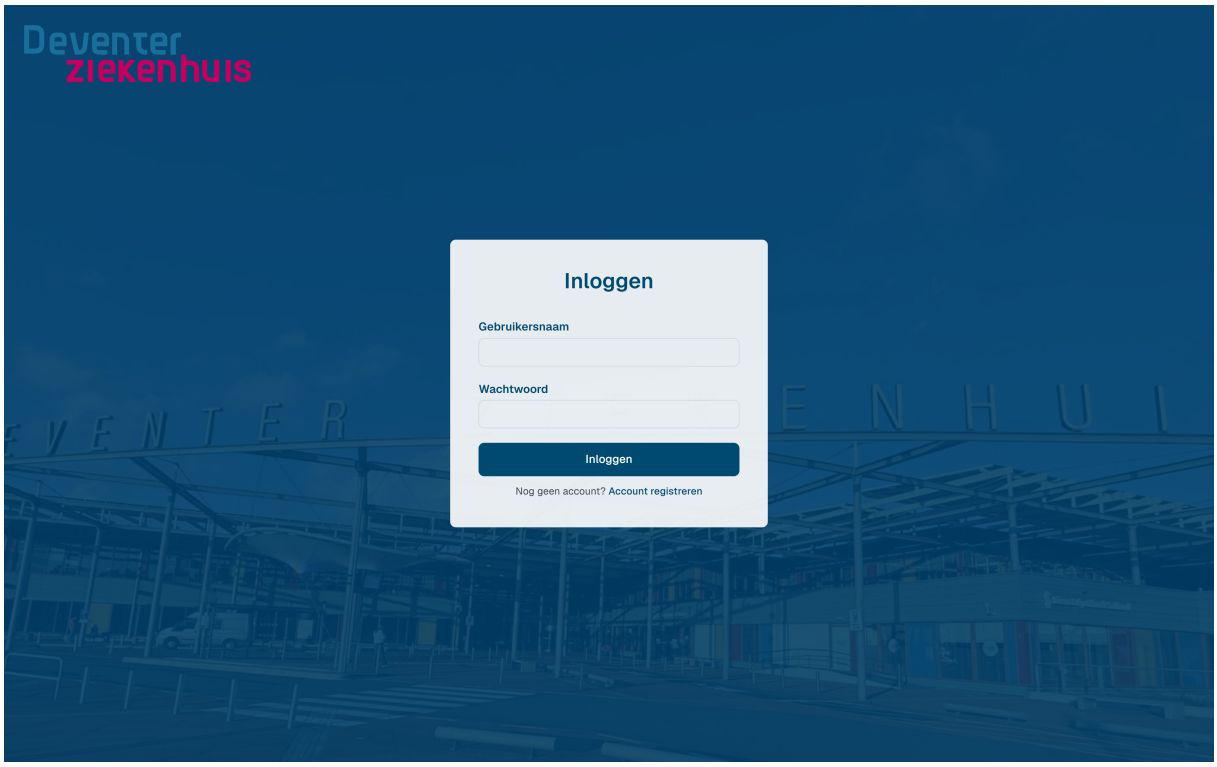


Figure 9: Login page

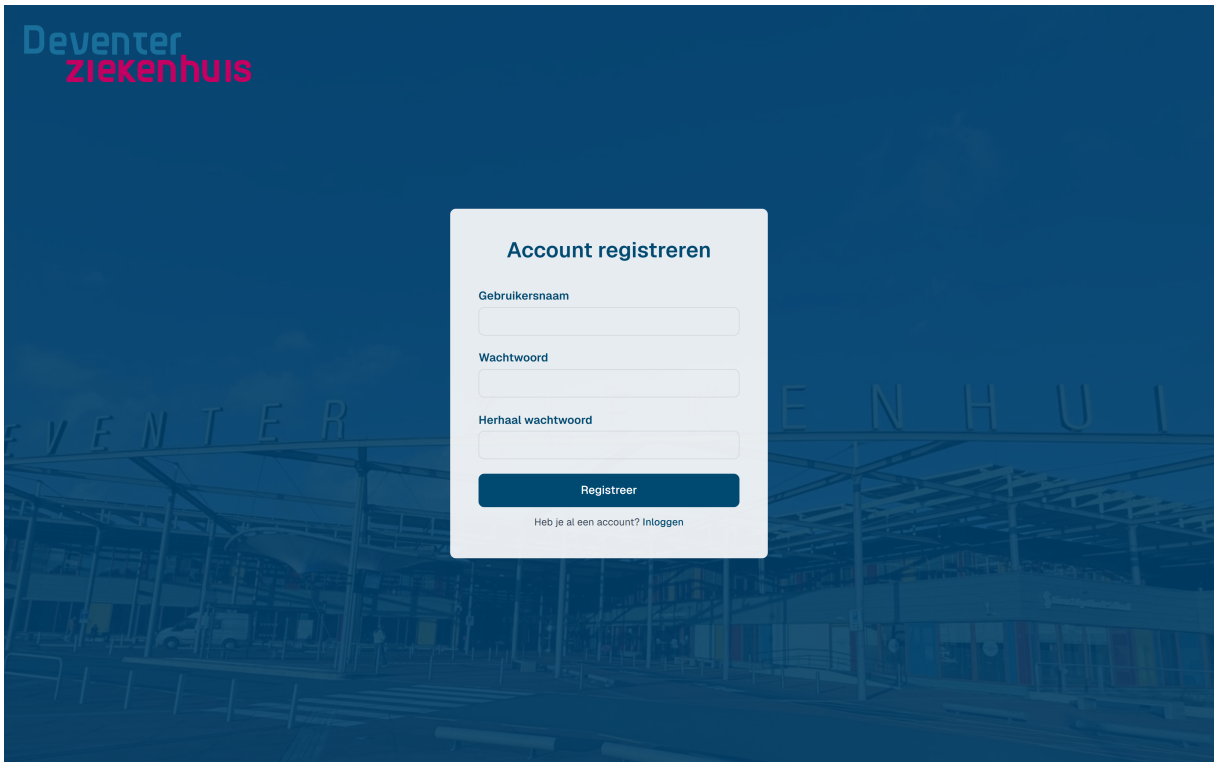


Figure 10: Registration page

4.2.3 Task overview

The task overview shown in Figure 11 is the main progress dashboard. This page gives users insight in the research jobs they have created. The page includes a table filled with all the jobs they currently have, including the name of the project they belong to. The status column shows the progress of the specific job. Under "Voortgang" the user can see how many rows of the data source have been processed. When they click on a table row, the website will navigate to the "Projecten" tab and open the corresponding project. The next chapter will discuss the project(s) page and their action buttons.

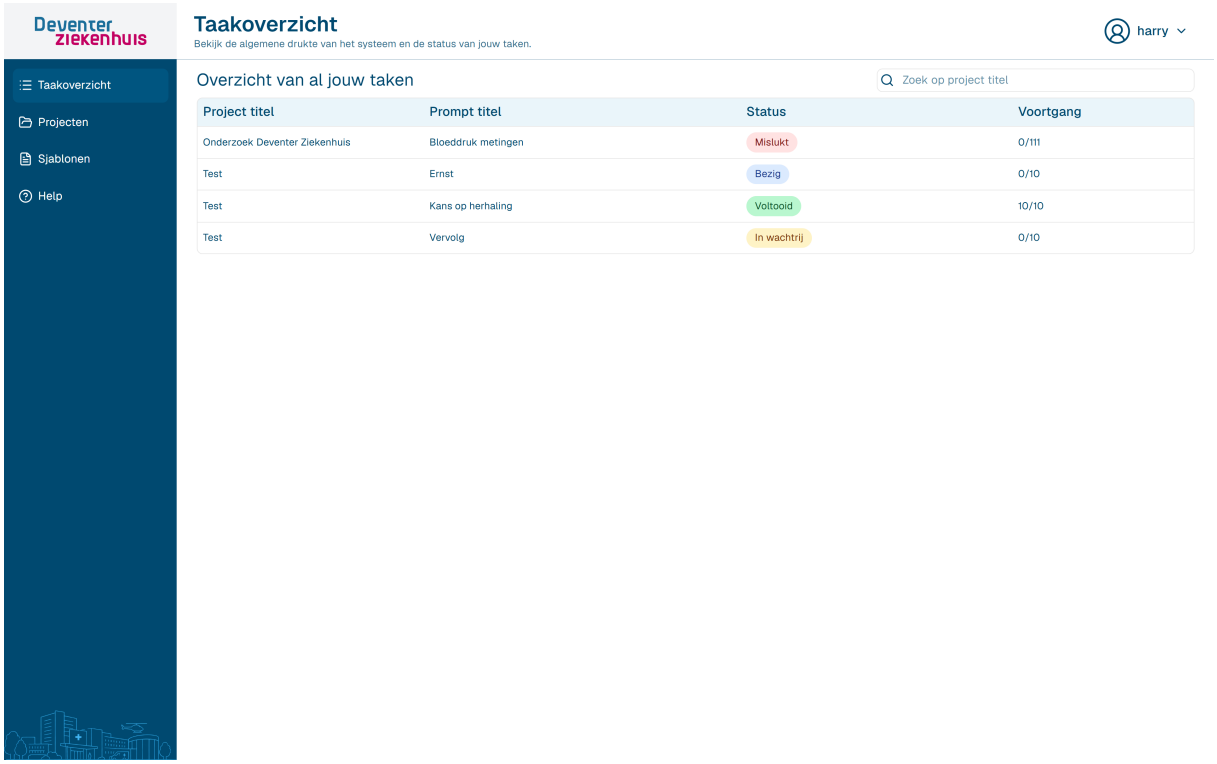


Figure 11: Task overview

4.2.4 Projecten

The projects page is the project management page which allows the user to see their own projects and execute certain actions. This page presents again a table containing the project title, amount of finished tasks of the project, the projects' status, last edited date and a column of action buttons. The possible actions are: Edit, run, pause, resume, cancel and delete. Editing makes it possible to rename the project if that is desired. The play, pause, and resume actions control the project's execution state. Play starts execution of all the jobs in the projects. Pausing and resuming do exactly what they are intuitively expected to do. Canceling a project terminates its execution. Running it again does start the project from the beginning. Clicking on a row in this table navigates to the project itself whose management page is shown in Figure 13.

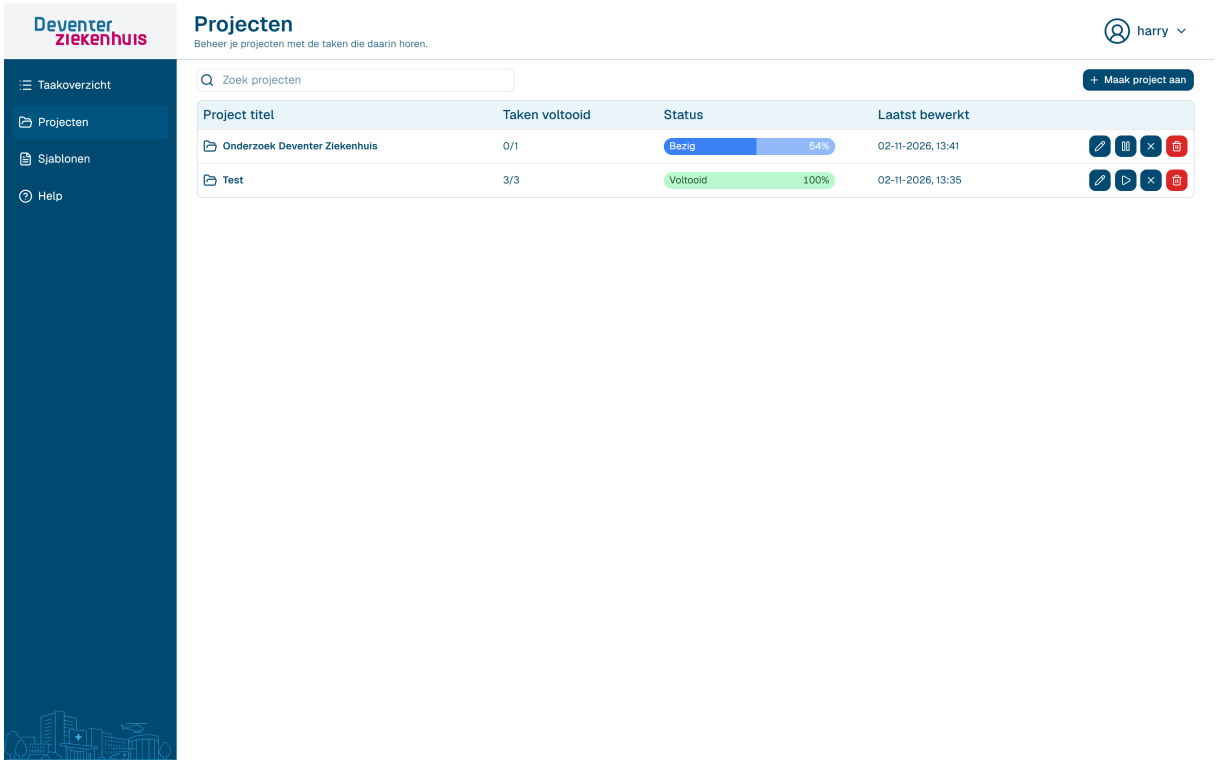


Figure 12: Projects overview page

On this page the user is presented with a table, this time containing the tasks/jobs that were created in this project. On the right hand side a CSV file can be uploaded. This file is processed and the columns are presented under "Selecteer kolommen", which allows the researcher to select which columns in the file he or she wants to use in their analysis. Another important part here is the ID column. This column determines what should be considered as the record id of a CSV row. Below this columns section the user can also select what range of rows they want to use in the project. After this a new prompt can be created to run on the imported data. This can be achieved by clicking on the "Voeg prompt toe" button which opens the new prompt creation popup shown in Figure 14

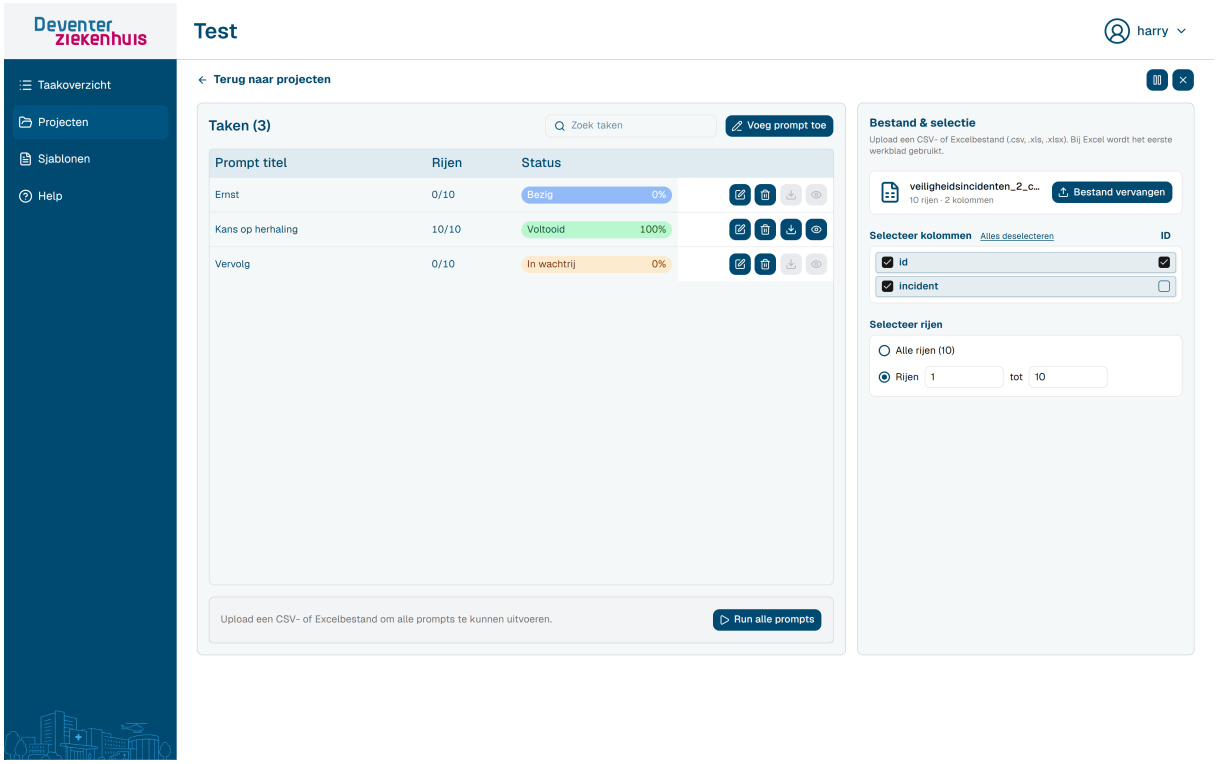


Figure 13: Project view

As can be seen below in the figure, a prompt involves several input fields. Obligatory are the prompt title, actual prompt and the output columns. The prompt title is just to be able to display the job in a compact way in the tables and to be able to recognize it quickly. The prompt itself is the instruction to the LLM. To allow for filtering on the output of this prompt, output columns are defined with a specific type. Take for example the prompt in the figure that asks whether the incident involved a fluid leakage. This just investigates one research question, which means one output column is needed, but in case the user has more than 1 question more output columns should be added to allow the system to output the results. To make this a boolean output, in the image the 'boolean' option is selected among the available types, which are 'string', 'integer' (whole number), 'number' (decimal number), 'boolean' and 'enum'. The enum type allows categorical values, such as the risk level or likelihood of recurrence of an incident. At the lower part of the popup a simple CSV preview is displayed, showing the researcher what he is working with. At the bottom right there is a button "Opslaan als sjabloon" which provides the possibility to save the prompt for later use, which includes the title, prompt text and output columns together with their types. This can then be imported in the "Laad sjabloon" dropdown on top, which allows the user to use such a saved prompt again. At the bottom left there is a checkbox "Deel sjabloon met alle onderzoekers" as well, which provides the possibility to enable other researchers to use this prompt. The last notable part is the "Gebruik output van geslaagde prompts als input" button, which brings the user to another popup to select outputs from earlier prompts and use them as an input for this prompt.

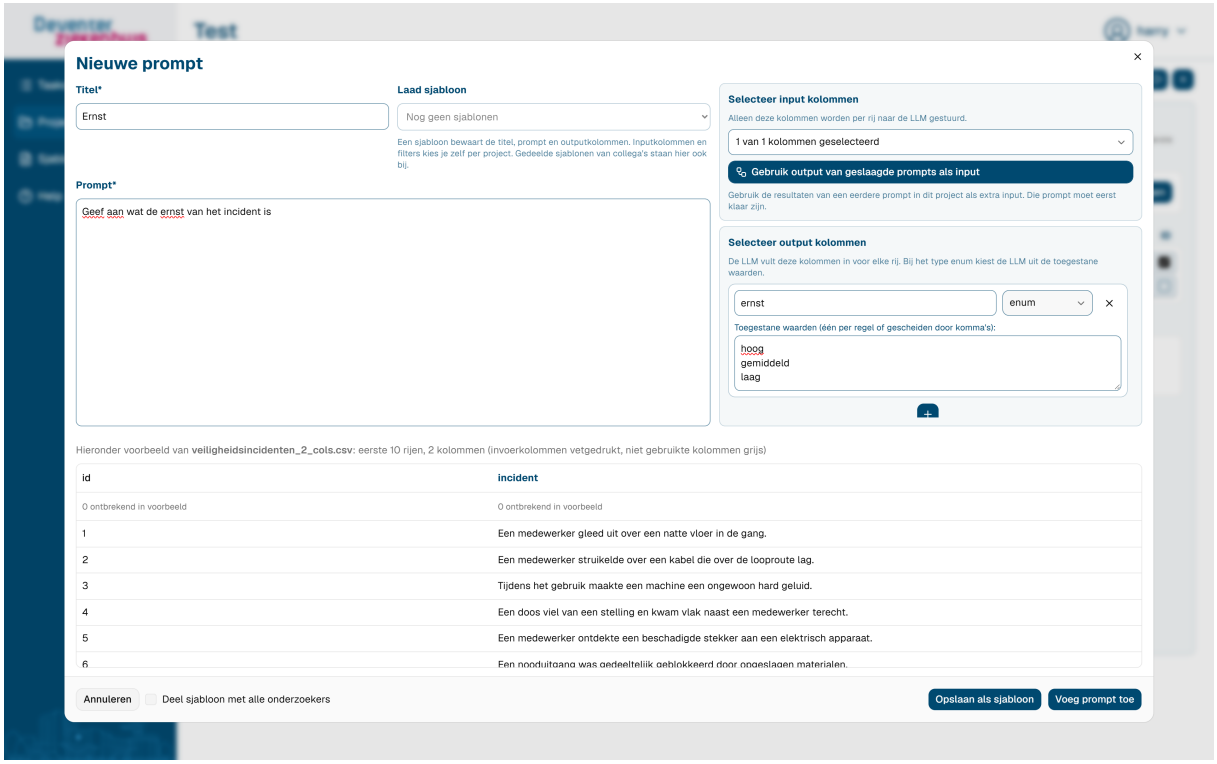


Figure 14: New prompt popup

In the figure below the popup for selecting outputs from other prompts as input is presented. On the left hand side there is a list of other prompts in the project which the user can unfold. Unfolding it reveals all the output columns that the prompt had specified in it. The researcher can then select those and on the right hand side specify possible filters. Filtering for example a boolean output columns on only the rows that had 'true' as output, will restrict the input of the new prompt to those rows.

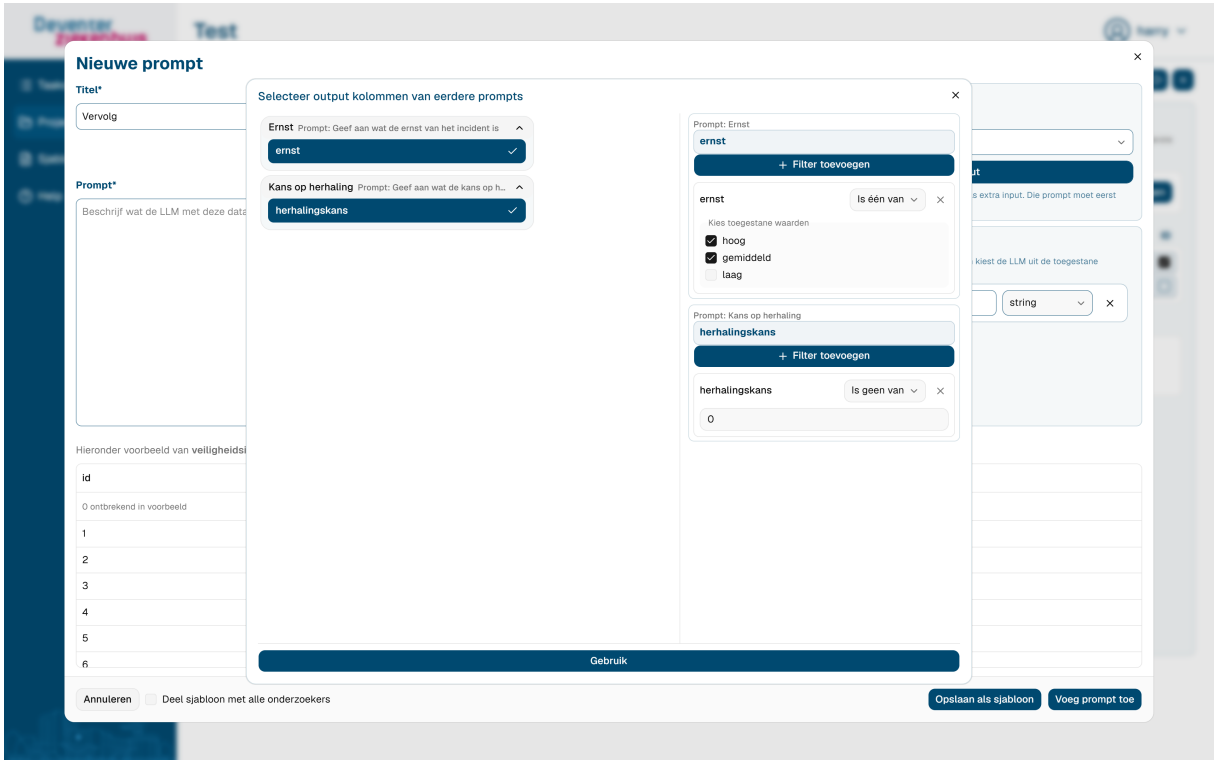


Figure 15: Earlier output columns selection with filters

From the jobs table of a project (Figure 13) next to downloading the output CSV, users can view the output of a completed job. This opens a popup with a view of the record id (taken from the id-column that was specified) together with the output column(s) that the prompt produced. The researcher can specify a range of rows to view in order to check the output, and download the CSV here as well.

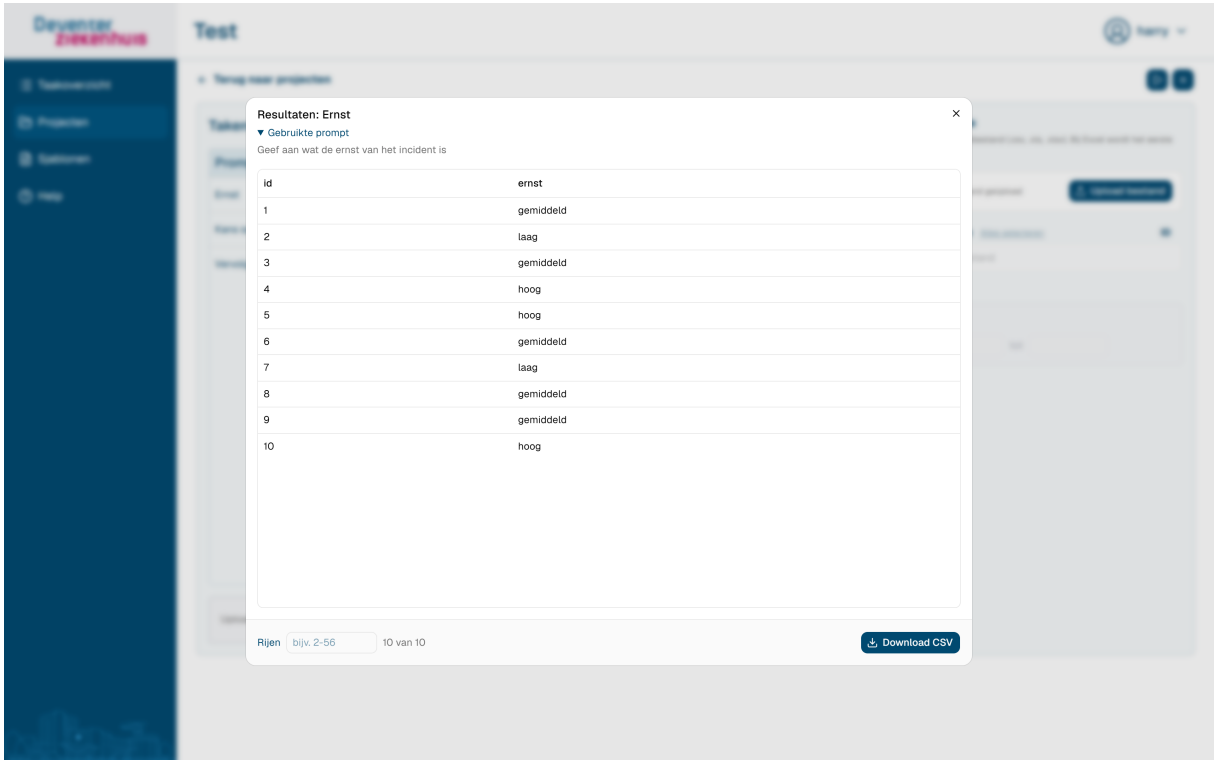


Figure 16: Output viewer

4.2.5 Sjablonen

In order to manage the prompts that researchers save for later use, a prompt templates (sjablonen) page was added. On this page researchers can manage their prompt templates, and see the ones added publiced others. They can add new templates and delete old ones. Below, the page is shown together with a picture of the popup for creating a new prompt template.

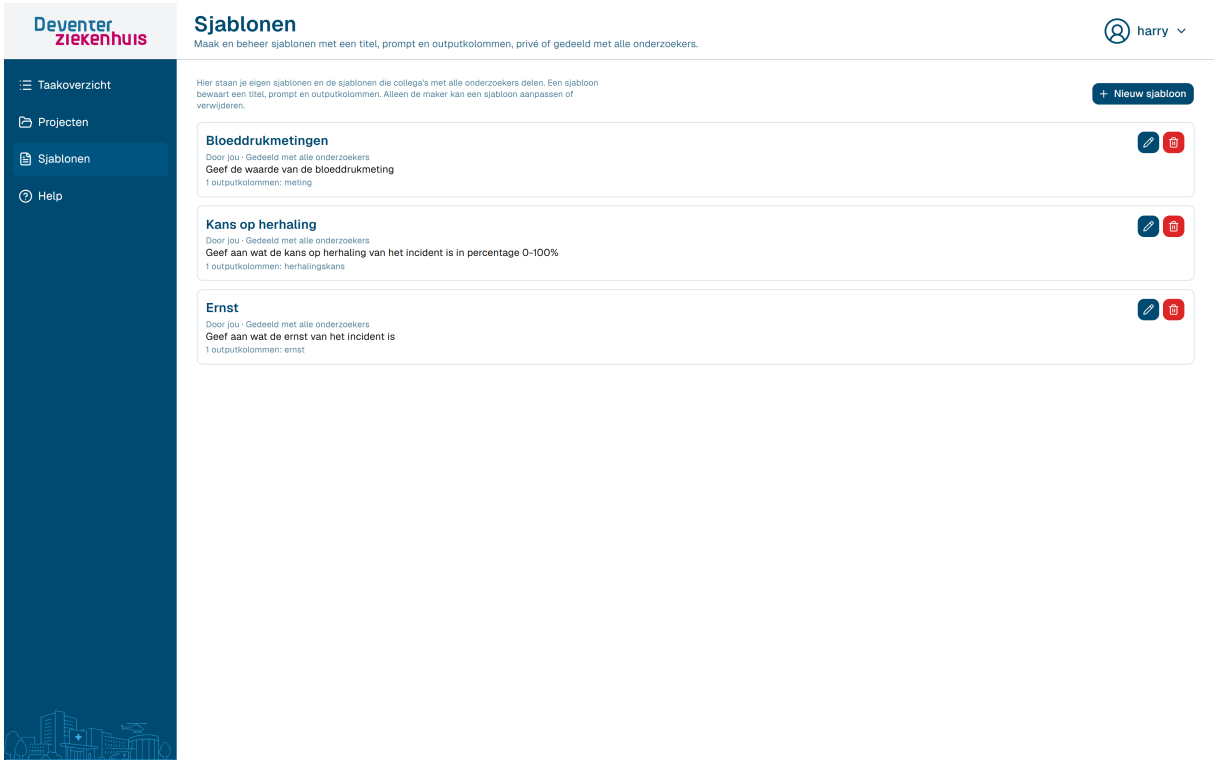


Figure 17: Prompt template page

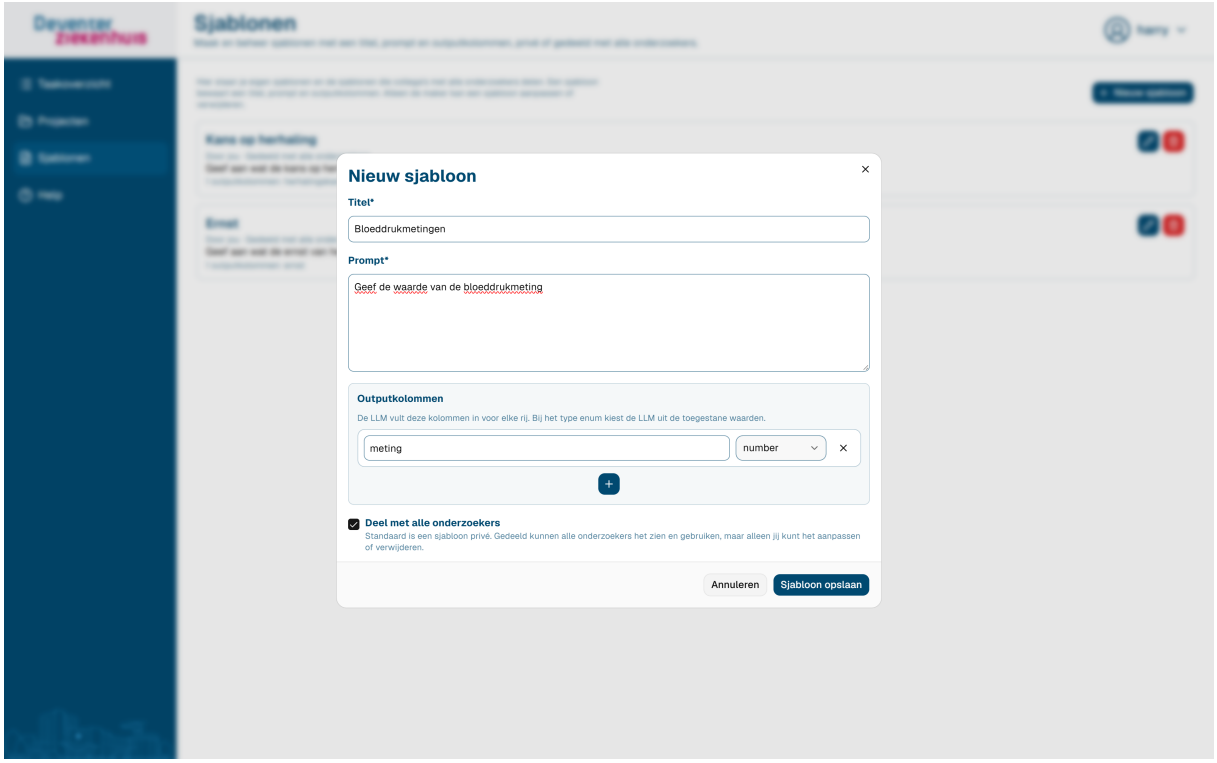


Figure 18: Add new prompt template

4.2.6 Help

For users to be able to navigate through the application smoothly, a help page is added. As shown below, this page presents several paragraphs of explanation on different pages of the application. This page should provide all the necessary information to use all functionality of the system.

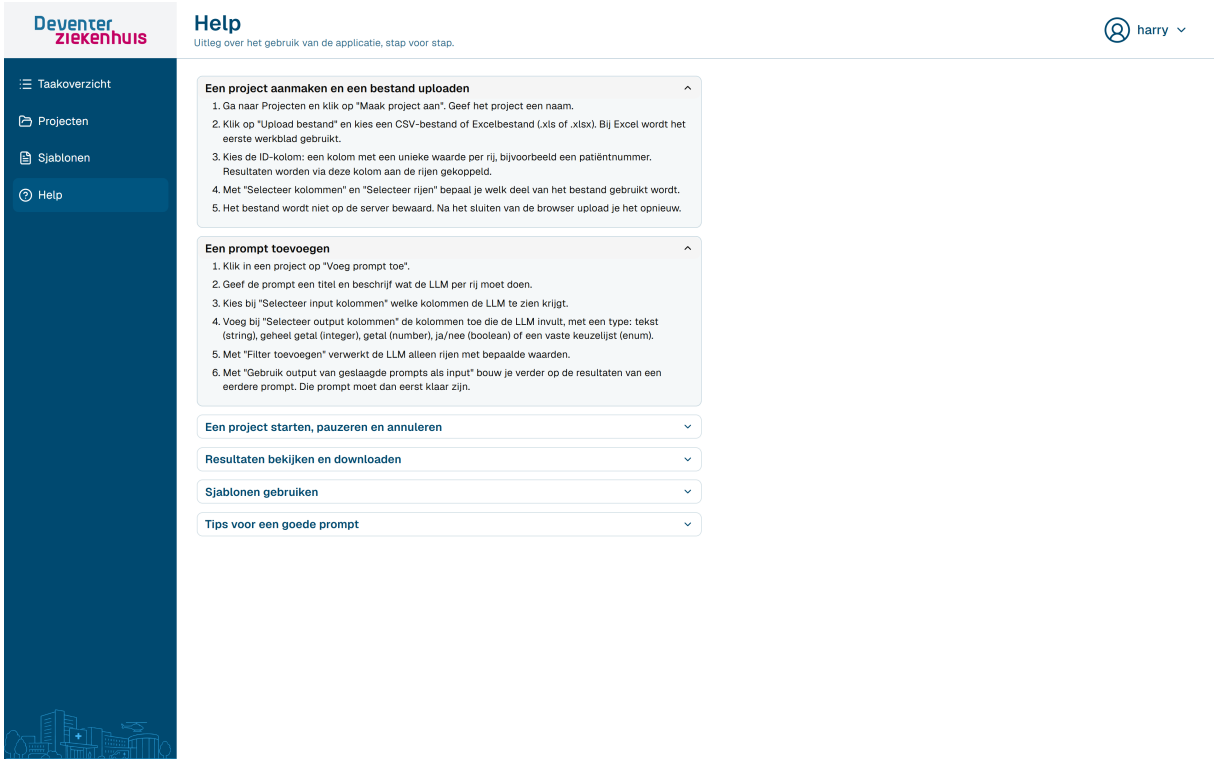


Figure 19: Help page

4.3 User and developer guide

The user guide is integrated into the Help page and follows the actual workflow: register or log in, create a project, upload a CSV, select the identifier and input columns, create a prompt, define typed output columns, test a small row range, run the project, monitor progress, inspect results, and export the result CSV. The guide explains that a patient identifier is a technical row key and must be removed or pseudonymised before sharing. It also explains that source data must be uploaded again after a pause or resume, because it is not stored by the application.

The developer and deployment guide is maintained in the repository README. It describes the Python and Node prerequisites, virtual-environment setup, frontend build, backend startup, environment variables, model endpoint, worker count, batch size, token settings, and API documentation. The hospital handover procedure is: install Python and Node on the isolated Linux machine, install the Python and npm dependencies, configure `backend/.env` from `.env.example`, ensure Ollama and the selected model are available locally, build the React assets, start the FastAPI service, and verify registration, a artificial CSV run, pause/resume, result export, and the absence of the debug router. This makes installation repeatable without requiring the project team to access the hospital network.

5 Testing

5.1 Test Plan

Testing was organized around the following research questions: does the implementation behave correctly, can a researcher use the main workflow, and how do model and temperature choices affect the result? Unit tests were written as well, these covered authentication, database operations, project/job lifecycle transitions, scheduling, retries, CSV validation, and API responses. There was also a model evaluation test which used the same binary classification dataset and prompt for multiple runs.
The temperature test was run multiple times at each temperature and the observations were collected in Figures 20 to 22. All the benchmarks used the `tc_test_good.csv`, containing 200 realistic artificial contact summaries (in Dutch): 144 positive labels and 56 negative labels. The ground-truth column was `tc_gedeclareerd_terecht`; the model had to return one integer label per `patient_id`. This dataset is a close copy of the dataset researches actually use at the DZ. The experiment evaluated temperatures 0.0, 0.2, 0.4, 0.6, 0.8, ..., until 2.0 for the configured models. Each result was then compared with the truth column, whilst identifying wrong output formats and evaluating the models accuracy. Note, again, that results are a controlled technical benchmark with realistic artificial data, not an actual clinical validation study.

5.2 Unit and integration testing

The backend test folder checks behaviour and interactions between the API, database, scheduler, and worker. In particular, invalid identifiers, malformed output, duplicate rows, failed requests, retries, pause/resume, and job dependencies were tested. These tests protect the requirements that already completed work is retained and that malformed model output does not silently become a successful result.

The tests were run with Python 3.13.9, pytest 9.1.1, and coverage.py [27, 28]. The package collected 29 tests and all 29 passed. The coverage was 81% (1,471 of 1,827 statements). Table 9 displays each test file to the main classes and subsystems it exercises and reports the relevant coverage. If changes were to be made to the system these tests can serve as validation of the system.

Test file	Main classes/subsystems covered	Relevant measured coverage	Result
`test_auth_refresh.py`	Authentication service, JWT security, token model, authentication routes and protected-endpoint dependencies	Auth/security/token modules: 100%	10 passed
`test_database_full.py`	SQLite database helpers, users, prompts, projects, jobs, results, refresh tokens, serialization, validation, and cascading deletion	Database module: 70.6%; model modules: 79.5–100%	2 passed
`test_full_project_workflow.py`	End-to-end template, project, four-job execution, scheduler completion, status reporting, and merged results	Project service: 80.7%; prompt service: 95.7%	1 passed
`test_parallel_jobs.py`	Parallel job creation, scheduler work batches, row processing, and persisted job results	Scheduler: 100%; jobs router: 96.6%	1 passed
`test_project_api.py`	Project/job/prompt API routes, CSV validation, duplicate IDs, temperature input, progress, access isolation, and deletion	Project router: 93.8%; jobs router: 96.6%; prompts router: 96.0%	11 passed
`test_scheduler_full.py`	Queue dependencies, batching, retries, failure, pause/resume, cancellation, invalid sources, and state cleanup	Scheduler: 100%; job model: 95.2%; project model: 95.7%	4 passed
`test_temperature.py`	Model-temperature benchmark and chart generation; this is an evaluation script rather than a pytest unit-test module	Evaluated separately from the 81% unit-test total	Not collected

Table 9: Python test files, covered subsystems, coverage, and results.

There was a low worker module coverage (18.3%). This is the case because tests drive the scheduler with deterministic work completion rather than making live LLM requests. This was intentional because live endpoint calls would make unit tests slow, non-reproducible, and dependent on hospital infrastructure. Worker integration with a real model remains a deployment and performance test.

5.3 User acceptance testing

5.3.1 Testing plan

To conduct a small user testing study a test plan was constructed. It is a black-box functional test, meaning that the system is evaluated through its observable behaviour and outputs without inspecting its internal implementation, following the terminology of ISO/IEC/IEEE 29119-1 [26]. The complete test plan is included in Appendix I, and the collected participant answers are included in Appendix J. Each session is planned to last approximately ten minutes, participants may stop without giving a reason, and only the supplied test data may be used. Note that the participants are researchers at the DZ, these are the end-users of the system, but this means there is a small sample size.

The participant is asked to complete the following tasks independently:

1. create an account and log in;
2. open the task overview, find an existing task, and inspect its status and progress;

3. create and open a project;
4. create a prompt, select the required CSV columns, configure the output type, and save the prompt;
5. upload the supplied test file, select its unique identifier column, test the prompt on at most ten rows, and inspect the results;
6. start an analysis, follow its progress, and pause and resume it when those controls are available;
7. inspect results per row, search by an identifier, and export the results to CSV; and
8. save a prompt as a template, find it again, and check whether it can be reused.

After the tasks, the participant is asked about the general experience, including whether the starting point, navigation, and interface structure were clear. Further questions address project creation, prompt configuration, input and output options, CSV upload, prompt testing, progress monitoring, result presentation, and CSV export. The final questions ask which task was most difficult, which buttons or texts were unclear, which functions were missing, what should be improved first, and whether the participant would use CLINI in their work.

5.3.2 User acceptance and demonstrations results

The user acceptance test and client demonstrations show that CLINI indeed provides a usable workflow for researchers who need to process CSV files with the local LLM. Participants were able to create projects, configure prompts, upload and test data, run analyses, inspect results, and export the output. The feedback also confirms that the application is a clear improvement over the earlier workflow with the script because it brings prompt configuration, execution, and result inspection together in one interface. All participants indicated that they would be willing to use CLINI in their work. Formal clinical acceptance remains the responsibility of DZ and must be performed with approved, de-identified records.

Strengths of the application

- The interface is clear, accessible, and easy to use. The workflow, including project creation, CSV upload, prompt configuration, testing, and export, can be completed without any extensive technical knowledge.
- The application combines several separate activities in one overview. Researchers can configure prompts, select input columns, run analyses, inspect row-level results, and export the output from the same project.
- Prompt testing on a limited number of rows makes it possible to check a prompt before processing a complete dataset. The result inspection control was really appreciated because it allows users to review the generated output per row and quickly assess it.
- Reusable prompts, explicit id column selection, progress feedback, and pause and resume functionality support a reproducible workflow for larger analyses.
- The application overall is considered a significant improvement over the previous implementation and all participants wanted to use it in their work.

Weaknesses and improvements of the application

- Progress reporting is too chucky for some datasets. The progress bar may remain at zero for several minutes and then jump to 50 or 100 percent, making it difficult to determine whether the analysis is still running.
- Input columns cannot be selected for inclusion in the output CSV. Users may need to join the generated results with the original data manually in R or Python.
- Some elements are not clear by design. Specifically, users may not know that the CSV must be uploaded before configuring a prompt, and the behaviour of using the output of one prompt as the input of another prompt requires better explanation within the app.
- Error feedback and recovery are incomplete. A failed task does not always provide a clear reason, and the action for restarting a failed task is not immediately visible.
- The current Excel workflow uses only the first worksheet, and pausing is not available from every task view. These limitations reduce flexibility.

5.3.3 Evaluation of results and advise

The results in Appendix J show that the main workflow was understandable, but that progress feedback was sometimes too coarse. The current scheduler processes rows in configured batches (3.3.5), and it was already noted that smaller batches provide smoother progress feedback while larger batches reduce overhead (3.3.5). The batch size could therefore be adapted to the dataset: smaller batches for short analyses and larger batches for large files. The current progress formula (Equation 3.3.5) could also be complemented or replaced by an estimated remaining time based on completed batches. Such an estimate should be presented as approximate because model response times vary. Other considerations are preserving clear error messages, showing when a job is active, keeping the maximum prompt-test size, and validating these changes with more participants and larger datasets using the test plan in Appendix I.

5.4 Testing Mistral against other models

The temperature evaluation used the `test_temperature.py` script, the `tc_test_good.csv`, and the same Dutch prompt. This follows the setup described in Section 5. The experiment simply compares the tested models and settings, but note that it does not establish clinical superiority at all, as also discussed in Section 6. Additional results about performance for Mistral were collected and evaluated. The goal of this test was to show that mistral in this setting can be (almost) outperformed by (even older) other models. Running the test resulted in the following:

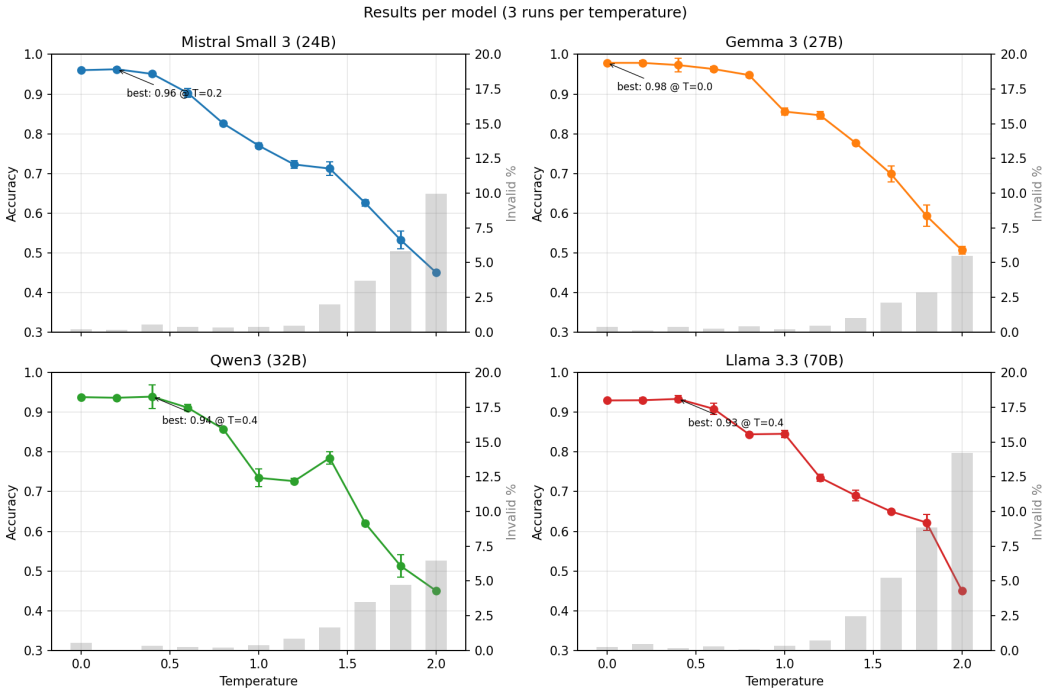


Figure 20: Accuracy vs temperature for the testing models.

Figure 20 and the heatmap in Figure 21 show that most models perform best with lower temperatures (0.0-0.4). Temperature controls the randomness of token sampling [23], so lower values make the model more likely to repeat its highest probability continuation. This is the expected behaviour for a classification task: the desired output is constrained to a small set of labels, so additional sampling variation is more likely to introduce an incorrect label or unwanted text than to add useful information. This also matches the system design, which instructs the model to produce a specific type of answer; see Section 3.3.7. The model temperature generally affects classification performance and should therefore be finetuned rather than assumed to have one universal optimum per model [23]. Also note that the used models for this test are about a year old, so newer models might outperform mistral even more. Nevertheless, temperature should remain a model-specific configuration because the curves and their variation differ between models.

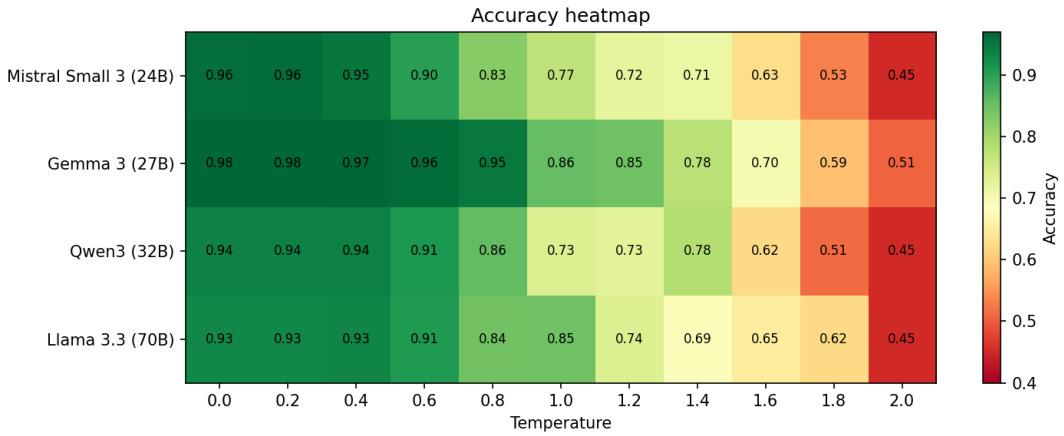


Figure 21: Accuracy heatmap by model and temperature.

Gemma clearly stands out in parts of the comparison. A possible explanation is that its multilingual

training and model design support Dutch text better than some of the other candidate models. Gemma 3 is explicitly described as a multilingual model with a long context window [37]. However, this experiment does not isolate language coverage from model size, instruction tuning, or sampling behavior. The language constraint was important in this project because the intended records are Dutch as mentioned in 2.1. The benchmark is therefore more realistic than a generic English classification example, although the artificial dataset and single task still do not represent the full variation of clinical research data. The output could also be biased by the way the prompt was constructed. A newer gemma model (or other newer models) could also be tested separately with a larger Dutch benchmark on hospital infrastructure to confirm.

The invalid-output chart in Figure 22 must be read separately from accuracy. A response can contain the correct classification but still fail the required format, while a syntactically valid response can still contain an incorrect classification.

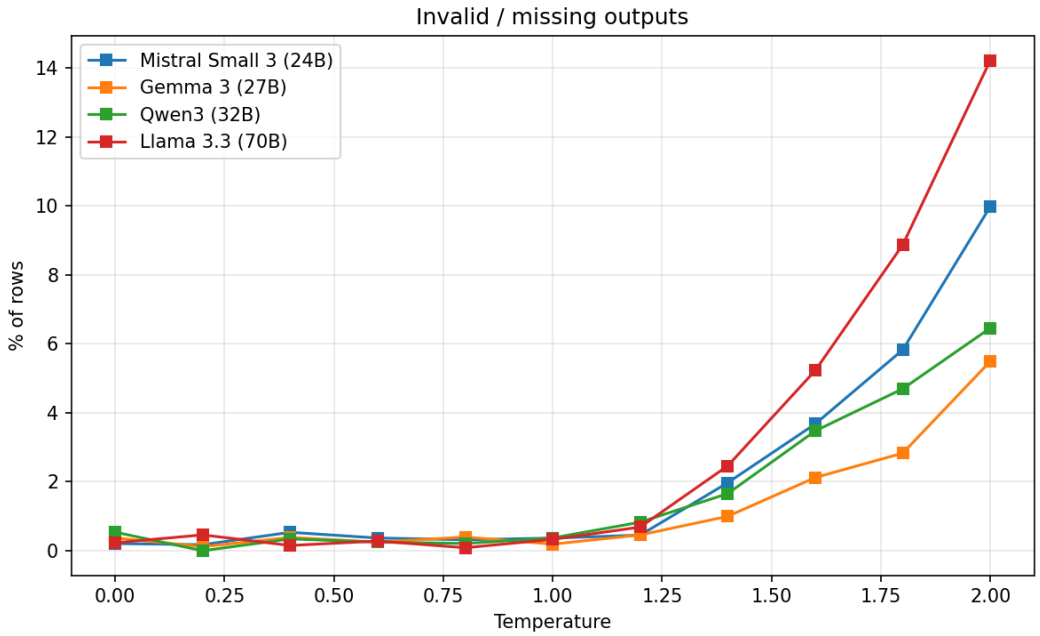


Figure 22: Invalid or missing output rate by model and temperature.

5.4.1 Evaluation

For the implemented workflow, a low temperature, preferably 0.0 or 0.2, is a reasonable starting point for structured extraction. Note that for some models a temperature around 1.0 was advised, and still only scored about 70 percent accuracy. Some hypotheses for this that still need to be tested are that the language was a constraint, or the user prompt with the system prompt were constructed improperly. The result of these tests are a configuration recommendation based on Figures 20 to 22, so not a claim that the same value is optimal for every prompt or model. Researchers are advised to test this again on a smaller range, inspect invalid outputs, and compare accuracy against approved ground truth before launching a full dataset run. They should also test on multiple prompts and experiment with newer models. The model and temperature remain configurable so it is advised that the DZ also re-evaluates these observations with actual clinical data and the correct setting. The bottom line is that Mistral might not be the best fit at all, which was claimed by the client at first.

5.5 Extra results Mistral

5.5.1 Local inference throughput and capacity

The deployed `mistral-small:24b` model was tested on the RTX 6000 Ada using the hospital's dataset. With 20 workers and an 8k context window, the test completed successfully and used approximately 42 GB of the available 48 GB of GPU memory. The result is specific to the tested dataset, model, hardware, Ollama configuration, and batch size; it is not a general capacity guarantee. The batch size must be configured so that the complete request, including the prompt and generated answer, fits within the 8k context window.

5.5.2 Confirmation of the memory estimate

The measured usage is consistent with the memory estimates in Section 3.3.7: approximately 14 GB for the model weights and approximately 27 GB for the KV cache of 20 parallel 8k slots, with the remaining memory used by runtime overhead. This relationship between parallel requests and KV-cache memory is also described in the LLM-serving literature [22] and the Ollama documentation [32].

Measurement	Observed value	Interpretation
1 worker tokens test	About 58 tokens per second	This is consistent with the expected value.
20-worker stress test	20 workers, 8k context, approximately 42 GB of 48 GB GPU memory	Successful run on the hospital dataset; approximately 6 GB of GPU memory remained available.
Processed rows	About 10 row/s measured	This is reasonable with 20 workers.

Table 10: Observed Mistral capacity test and corresponding memory estimate.

5.6 Large limitations to the evaluation

The scope of the evaluation of these tests was limited by both the available infrastructure and the nature of the project. Since this was developed for deployment in a hospital environment, testing very heavily depended on the available hardware and resources. This limited the extent to which large-scale performance, accuracy, and concurrency tests could be conducted. This will be elaborated on in the next sections.

6 Discussion

6.1 General view

The final prototype successfully transformed the original existing Python script into a workflow for information extraction. Researchers can write and store prompts and typed output fields, run jobs, monitor progress, and export results without directly interacting with the LLM infrastructure. The scheduler, batching, retry handling, and persistent metadata address several limitations of the original workflow.

The prototype also shows that an on-premise LLM can be made accessible to users while keeping source data within the hospital environment. However, CLINI remains a research prototype and analysis-support tool, not a clinically validated information-extraction system. The project demonstrates technical feasibility, but clinical correctness, generalisability, governance, and long-term operational use require much more further investigation.

6.2 What worked

The separation between the persistent data and volatile runtime data supports the privacy requirement that CSVs should not be stored unnecessarily. The scheduler handles dependencies, batching, pausing, resuming, cancellation, and retries in a consistent way. Invalid model responses can be retried without discarding already completed work.

Typed output definitions also improve the reliability of the interface between the LLM and the application. They ensure that results have the expected structure and make invalid values visible. However, schema validation only confirms the format and type of an answer and it does not confirm that the extracted information is clinically correct of course.

The model and memory tests showed that temperature and concurrency depend on the chosen model and hardware. For the deployed `mistral-small:24b` model, up to 20 concurrent workers could be used on the 48 GB GPU under the tested conditions. This is a useful starting point, but not a universal optimum for other models, prompts, context sizes, or workloads.

6.3 Limitations of the research and evaluation

The research and testing could have been broader and more extensive. The project focused mainly on developing a usable prototype, which limited the extent of the literature review, model comparison, and evaluation of the healthcare context. More structured interviews with researchers, data managers, clinicians, privacy officers, and IT administrators would have provided a broader and more complete product.

The model evaluation was very limited. We did not really test a sufficiently large variety of prompts, prompt structures, context window sizes, batch sizes, input lengths, temperatures, or even on the newest models. Prompt design was mainly treated as an implementation detail rather than as an experimental variable. Different prompts can produce substantially different results even when the underlying model remains unchanged.

The comparison also included only a limited number of models. Indeed newer models may provide better extraction quality, context handling, speed, or resource usage. More recent models, different model sizes, and different quantisation settings should therefore be evaluated before selecting a model for long-term use. Since testing the model was only possible on the hospital infrastructure this was a big limitation. On this there is much more to research and investigate on.

A more complete evaluation should use a fixed and representative benchmark and measure

accuracy, precision, latency, throughput, GPU memory usage, etc. The interaction between model choice, prompt design, context length, temperature, batch size, and concurrency should also be investigated more thoroughly.

artificial data was necessary because real patient data could not be used in the project environment because of privacy constraints. However, artificial data cannot fully reproduce the abbreviations, incomplete sentences, missing context, and ambiguity found in real clinical notes. The results should therefore be interpreted as an initial feasibility assessment, not as evidence of clinical reliability. Also on this part more research must be done before using and deploying the product.

The deployment test confirmed that the workflow could run on the intended hospital machine. More repeated tests with larger datasets and realistic concurrent users would be required to draw conclusions on scalability.

6.4 Technical and methodological lessons

A valid LLM response can still contain an incorrect extraction or a hallucination. Future versions of CLINI should make this visible and allow researchers to review each output alongside the original source text. Model, prompt, and configuration versions should also be recorded to improve reproducibility.

The privacy-preserving design introduces usability trade-offs. For now, source files are removed from runtime when a project is paused, meaning that researchers must upload them again when resuming. This makes the workflow less convenient.

The project also showed the value of deployment feedback. The conducted test on the hospital revealed practical issues that were not obvious during development. Earlier and more frequent deployment sessions would likely have identified these sooner.

6.5 Responsible use and handover

CLINI makes analysis more accessible, but responsibility must remain clear. Researchers are responsible for reviewing and interpreting results, and the DZ is responsible for approving datasets, prompts, models, access rights, retention policies, and result sharing before operational use.

Hallucinations and output bias remain a significant risk because a well-structured answer may appear reliable even when it is wrong. The implemented typed schema's, retry handling, provenance metadata, and human review reduce this risk but can never fully eliminate it. CLINI should therefore not immediately be used as a diagnostic system.

Before final deployment, the hospital should define review procedures, model and prompt approval processes, access controls, logging, backup policies, and incident procedures. Sharing the system with another hospital would additionally require a separate legal basis, security assessment, deployment procedure, and license review.

6.6 Future work

The next step should be a controlled evaluation using approved, anonymized clinical records and domain experts. This evaluation should compare a broader range of prompts, context window sizes, models, temperatures, batch sizes, and concurrency levels using the same fixed benchmark. Results should be validated on unseen data to reduce the risk of overfitting prompts to the test set.

Future technical improvements include stronger provenance, human correction of individual outputs, version tracking for models and prompts, improved output validation, monitoring of

repeated failures, and role-based access control if the system is connected to the hospital network. Human in the loop workflows and domain-specific preprocessing could also be investigated.

Overall, the project achieved its primary goal of demonstrating a reusable on-premise LLM workflow for healthcare research. Its main limitation is that the research and evaluation were not extensive enough to establish clinical validity. More thorough testing with different prompts, context windows, models, input conditions, and newer LLMs is necessary before a long term operational version can be selected.

7 Conclusion

In this work CLINI was presented. CLINI addresses the central problem identified at the start of this report: researchers need to extract structured information from sensitive clinical text without depending on technical staff or cloud services. The resulting system combines a local LLM workflow with typed outputs, job scheduling, progress monitoring, and reusable prompts. Its architecture is deliberately shaped by the isolated machine, privacy requirements, limited GPU memory, and the needs of non-technical stakeholders.

The evaluation indicates that low temperatures are a sensible starting point for this binary task, but also shows why model and deployment measurements are necessary. Deployment and testing at the DZ showed that CLINI can serve as a promising starting point for further research with LLMs: it provides a workable bridge between local language-model inference and the practical needs of medical researchers. Further validation, human review, and refinement with approved hospital data remain necessary before drawing conclusions.

8 Project Management

The project applied a hybrid agile approach. An initial analysis and planning phase was combined with iterative development in weekly sprints. During the analysis phase, the client's existing workflow was researched, the requirements and constraints were identified, and the overall system architecture and report structure were established. This gave the project a shared direction before implementation started.

8.1 Justification of approach

A hybrid approach fitted this project because the requirements were only partly settled by the client. The client at Deventer Hospital knew the problem they wanted solved and had a clear idea of the intended workflow, but most requirements were classified as *should have* or *nice to have* rather than *must have*. The general direction was therefore clear, while the exact scope remained open to discussion and only became more fixed during the first weeks of the project. The supervisor also gave extensive feedback and technical guidance on architecture, hardware and model constraints, and the best approaches to follow, which regularly changed how the team approached a problem. A fully plan-driven method would have locked in requirements that were still evolving, whereas a fully agile method would have lacked the shared foundation needed to make sensible technical choices early on.

Development took place in sprints of approximately one week. Requirements and priorities were revisited whenever demonstrations, supervisor feedback, or technical findings changed the understanding of the problem. A summary of the sprints is included in Appendix E.

The team met weekly with both the client and the supervisor. The client meetings were used to validate the workflow, clarify requirements, demonstrate progress, and collect usability feedback. Because many requirements were optional, these meetings were also where priorities were agreed on: the client indicated which *should have* and *nice to have* features mattered most in practice. Before these meetings, the team held short planning discussions to prepare or reprioritise the board, so that it arrived with a shared overview of completed, in-progress, and blocked work. Decisions from the meetings were then translated directly into GitLab issues or changes to the sprint priorities. In addition, the team coordinated implementation work through branch discussions, merges, demonstrations, and shared report reviews.

8.2 GitLab planning structure

GitLab was used to manage the source code, epics, and issues, which were organised as a hierarchy. An epic represented a large part of the project, such as requirements and architecture, the web interface, the backend, testing, or documentation. An issue represented one concrete piece of work within an epic. Issues were written as user stories when they described user value (for example, enabling a researcher to configure an analysis) and as technical tasks when they described implementation work. Tasks could be listed inside an issue when a user story required several implementation steps. This distinction kept the project goals visible while still allowing the team to plan detailed engineering work. At the start of the project, the team collected the requirements, transformed them into user stories under the requirements and architecture epic, and then added technical tasks beneath the stories where necessary.

The issue board was the team's planning interface for the stand-up meetings. Whenever an issue was created, it was assigned to the project delivery milestone, given the appropriate priority, type, and workflow labels, assigned to the relevant team members, and placed in the iteration in which it was expected to be completed. This made ownership, timing, priority, and status explicit instead of leaving them in chat messages or personal task lists. User stories were moved to closed only

after the work covered by the story was complete, so technical tasks could remain visible on the board alongside user stories without being confused with them.

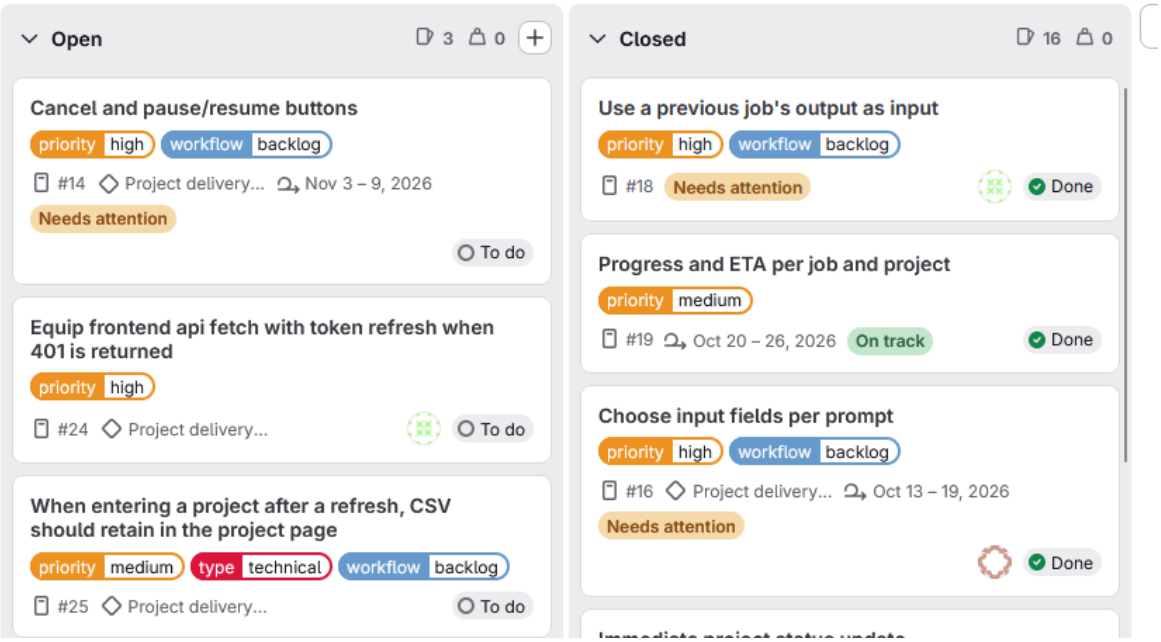


Figure 23: GitLab issue board used for day-to-day planning, assignment, and tracking of project work.

The project used milestones and one-week iterations together. The project delivery milestone represented the complete delivery period and provided the overall burndown view. Each weekly iteration represented one sprint and contained the issues planned for that week. At the start of an iteration, issues were selected from the backlog and assigned; during the week, their workflow status was updated on the board; and at the end, the team reviewed the completed work and moved unfinished issues to a later iteration when necessary. This gave the project a measurable weekly rhythm.

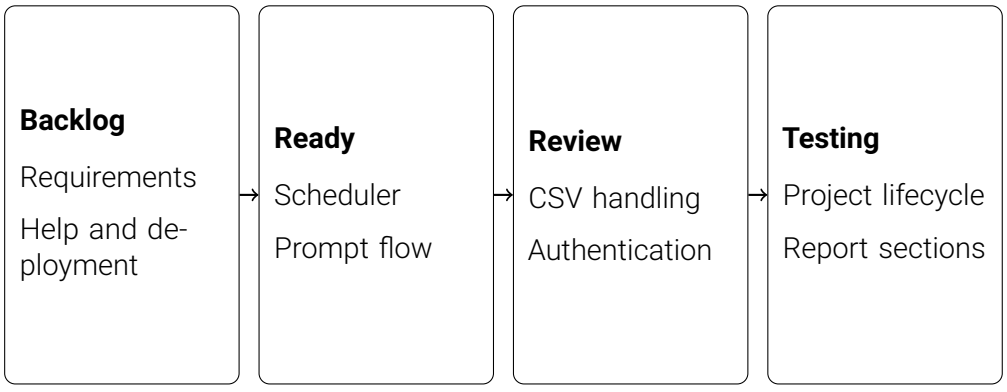


Figure 24: GitLab epic board showing the large project themes and their associated work items.

8.3 GitLab labels and epics

The label system made the status and purpose of an issue visible without reading its full description. The following label groups were used:

- **Priority:** `priority::high`, `priority::medium`, and `priority::low`, indicating urgency.
- **Type:** `type::bug`, `type::story`, `type::task`, and `type::technical`, distinguishing a user story from a bug, ordinary task, or technical task.
- **Workflow:** `workflow::backlog`, `workflow::ready`, `workflow::review`, and `workflow::testing`, showing the current stage: work began in the backlog, moved to ready when it could be picked up, and then passed through review and testing.

The labels were deliberately orthogonal, so one issue could, for example, be marked `type::story`, `priority::high`, and `workflow::review` at the same time. This made it possible to reprioritise an issue after a client demonstration or supervisor meeting without losing its history or its relationship to the larger feature.

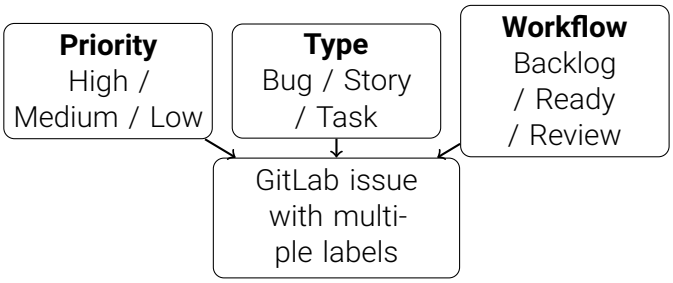


Figure 25: The GitLab label scheme used to classify and track issues.

The epic board provided a project-level view of progress: issues moved through the workflow while dependencies and blocked work stayed visible. A feature branch could be reviewed and merged while its epic remained open for integration tests, documentation, and demonstration, so a merged branch was never treated as proof that an entire feature was finished. Conversely, a bug discovered in a demonstration was added to the appropriate epic instead of being handled informally. The labels, milestones, iterations, and epics therefore served both as a planning instrument and as a traceable record of how client and supervisor feedback became implementation work. This allowed the team to show the supervisor not only the final software, but also the progress and decisions that led to it.

8.4 Sprint evaluation and adaptation

At the end of each sprint, the team reviewed the completed work together against the issues for that iteration. In this way they could evaluate whether the sprint objectives had been achieved and identify work that needed to be carried over to the next one. Unfinished issues were moved to a next iteration or returned to the backlog, depending on the priority and relevance.

When work could not be completed, the team considered the cause. This could include underestimated implementation effort, technical issues, or changes in priorities following client or supervisor meetings. These were taken into account when planning the next iterations.

This way of iterative planning process enabled the team to change its approach throughout development rather than having the initial planning as fixed. This was very useful because some requirements and technical decisions only became clear after implementation and demonstrations. As a result, sprint planning remained aligned with all requirements.

8.5 Risk and scope management

Risk and scope management were important because the project involved technical uncertainties and requirements that were not fully clear at the start. Risks included hardware limitations, model requirements, integration problems, and changes in client expectations.

These were addressed through early technical and personal investigation, guidance from our supervisor, and incremental development. This allowed limitations to be identified before they significantly affected the final implementation. Changing requirements was not necessary luckily.

Each risk was additionally scored on probability and impact ($P \times I$) before and after mitigation. The scoring scale, the full assessment and the resulting risk matrix are given in Appendix K. The assessment shows that the three highest-scoring risks (missing real patient data, the isolated machine, and hallucinated output) could be reduced but not eliminated, which is why the discussion treats clinical validation as future work (Section 6).

The initial identification and classification of requirements provided a basis for prioritization. Must-have requirements formed the foundation of the implementation, while optional features were reconsidered throughout development. This ensured that the project remained focused on the most important functionality.

8.6 Definition of Done and quality control

The team distinguished between completed implementation tasks and fully completed features. A merged feature branch did not mean at all that the feature was finished, as additional options, testing, documentation, or demonstration could still be needed for that function.

An issue was actually considered complete when the intended functionality had been implemented, relevant tests have passed, and the changes had been reviewed. If applicable, documentation that was already there was also updated. Features that require further validation remained open until the outstanding work was completed.

9 Contributions

Table 11 lists the main contribution of each team member. All members took part in the weekly meetings, the client and supervisor sessions, and the writing and proofreading of this report.

Team member	Main contributions
Harry van Dijk	Frontend foundation and most of the main user interface, including the project page, job and task tables, login flow, navigation, prompt creation flow, CSV and Excel upload support, row selection, and frontend error handling.
Jens Taken	All client communication, Backend and project infrastructure, including the initial API, database work, project and job models, LLM scheduling and worker logic, automated tests, setup documentation, and report maintenance.
Juras Mačijauskas	Frontend presentation and interaction details, including the result viewer, CSV preview, status bar, icons, tooltips, prompt templates, help page, status colours, and related styling.
Marlon Nelson	Frontend controls and project-editing behaviour, including input-column selection, row-range editing, CSV persistence, task-table and homepage changes, and pause and cancel error messages.
Ou-Xiang Chen	Frontend usability and project-page behaviour, including search bars, table sizing, process cards, project titles, row-selection behaviour, navigation persistence, and input/output column handling.
Sander Bertram	Backend routers and services for jobs, projects, batches, database access, schemas, and LLM scheduling; also error-handling simplification, Dutch translations, cleanup of unnecessary exceptions.

Table 11: Contributions per team member.

Contributions per component.

- **Requirements, MoSCoW analysis and client communication:** the full team, Jens was coordinating the requirements and documentation and all members contributing to the weekly client and supervisor discussions.
- **Backend API, authentication and database:** Jens and Sander, Juras was contributing to the integration and file-structure changes.
- **Scheduler, workers and job dependencies:** Jens and Sander, including the queue, worker execution, project state handling, retries, and dependency-aware processing.
- **LLM layer and model/temperature evaluation:** Jens and Sander, with the team contributing to the experiments, interpretation of constraints, and discussion of the results.
- **Frontend login, task overview, projects and prompt flow:** Harry, with contributions from Marlon and Ou-Xiang to the controls, tables, search, persistence, and project-page behaviour.
- **Frontend templates, result viewing, previews and visual design:** Juras, Harry, and Ou-Xiang, covering templates, tooltips, status presentation, CSV previews, result limits, layout, and navigation.

- **Deployment and testing at Deventer Hospital:** Jens and Sander.
- **Automated tests:** Jens and Sander both contributing to backend testing work and the full team using the tests and demonstrations to verify changes.
- **Report, poster and presentation:** the full team, with Jens maintaining the report structure and LaTeX source and all members contributing content, review, and presentation preparation.

10 Reflection

This section looks back at the working process. The technical results, their limits, and future work are discussed in Chapter 6.

10.1 Process

Throughout the project the GitLab iterations, issue board, and priorities were constantly aligned. The role provided structure while the team still shared responsibility for updating issues and completing work. Regular contact with the client was the most important factor for relevance. Demonstrations repeatedly changed project priorities, for instance towards shorter explanations in the interface, tests limited to ten rows, and the identifier column. Short iterations, client and supervisor meetings, automated tests, and end-to-end demonstrations together kept the prototype working throughout the project.

Feedback was followed up explicitly. The 30 September (Appendix H) discussion about agentic systems and LLM-as-a-judge was separated into two concepts: agentic multi-step execution remained future work, while LLM-as-a-judge was recorded as a possible evaluation method. The 1 October (Appendix G) request to edit individual rows for debugging became row-range selection, result viewing, and a ten-row test workflow. The 15 October (Appendix G) request to address ethical and legal responsibility became the privacy logging policy, the distinction between format validation and clinical correctness, and the handover requirement that the DZ defines human review before operational use.

10.2 What would be done differently

- **Deploy earlier.** The single on-site test in Sprint 7 produced the most useful findings, such as problems with job-output labels, filters on shared output columns, and editing input columns without re-uploading. Since remote access was impossible, an earlier session, or a closer replica of the machine at the university, would have shortened the feedback loop.
- **Writing the report continuously.** Parts of the implementation chapter and the help page were still unfinished in the last sprint. Writing each section in the sprint in which the feature was built would have spread the workload more evenly.
- **Define evaluation earlier.** The model comparison started only after the prototype worked. A small, fixed benchmark from the start would have shown trade-offs between models while design decisions were still open.
- **Record design decisions when they are made.** Several designs changed during the project (the task overview cards, the prompt page layout). Writing down the reason for each change would have made the design rationale easier to reconstruct.
- **More testing.** Of course, given the limited time in which this project was done, it makes sense that not too many tests could be conducted. It would have been great to draw a bigger conclusion about usability of the models in the product.

References

[1] S. T. Rosenbloom, J. C. Denny, H. Xu, N. Lorenzi, W. W. Stead, and K. B. Johnson. Data from clinical notes: a perspective on the tension between structure and flexible documentation. *Journal of the American Medical Informatics Association*, 18(2):181–186, 2011.

[2] S. M. Meystre, G. K. Savova, K. C. Kipper-Schuler, and J. F. Hurdle. Extracting information from textual documents in the electronic health record: a review of recent research. *Yearbook of Medical Informatics*, 17(1):128–144, 2008.

[3] Y. Wang, L. Wang, M. Rastegar-Mojarad, et al. Clinical information extraction applications: a literature review. *Journal of Biomedical Informatics*, 77:34–49, 2018.

[4] K. Kreimeyer, M. Foster, A. Pandey, et al. Natural language processing systems for capturing and standardizing unstructured clinical information: a systematic review. *Journal of Biomedical Informatics*, 73:14–29, 2017.

[5] N. G. Weiskopf and C. Weng. Methods and dimensions of electronic health record data quality assessment: enabling reuse for clinical research. *Journal of the American Medical Informatics Association*, 20(1):144–151, 2013.

[6] W. R. Hersh, M. G. Weiner, P. J. Embi, et al. Caveats for the use of operational electronic health record data in comparative effectiveness research. *Medical Care*, 51(8 Suppl 3):S30–S37, 2013.

[7] European Parliament and Council. Regulation (EU) 2016/679 (General Data Protection Regulation). *Official Journal of the European Union*, L 119, 2016.

[8] Netherlands Standardization Institute (NEN). *NEN 7510: Information security in healthcare*. Delft, The Netherlands.

[9] V. Menger, F. Scheepers, L. M. van Wijk, and M. Spruit. DEDUCE: a pattern matching method for automatic de-identification of Dutch medical text. *Telematics and Informatics*, 35(4):727–736, 2018.

[10] F. Dennstädt, J. Hastings, P. M. Putora, M. Schmerder, and N. Cihoric. Implementing large language models in healthcare while balancing control, collaboration, costs and security. *npj Digital Medicine*, 8:143, 2025.

[11] K. Singhal, S. Azizi, T. Tu, et al. Large language models encode clinical knowledge. *Nature*, 620:172–180, 2023.

[12] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, et al. Large language models in medicine. *Nature Medicine*, 29:1930–1940, 2023.

[13] M. Agrawal, S. Hegselmann, H. Lang, Y. Kim, and D. Sontag. Large language models are few-shot clinical information extractors. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[14] Y. Hu, Q. Chen, J. Du, et al. Improving large language models for clinical named entity recognition via prompt engineering. *Journal of the American Medical Informatics Association*, 31(9):1812–1820, 2024.

[15] A. Kong, S. Zhao, H. Chen, et al. Better zero-shot reasoning with role-play prompting. arXiv preprint arXiv:2308.07702, 2024.

[16] L. Y. Jiang, X. C. Liu, N. P. Nejatian, et al. Health system-scale language models are all-purpose prediction engines. *Nature*, 619:357–362, 2023.

[17] I. C. Wiest, D. Ferber, J. Zhu, et al. Privacy-preserving large language models for structured medical information retrieval. *npj Digital Medicine*, 7:257, 2024.

[18] Z. Ji, N. Lee, R. Frieske, et al. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12), Article 248, 2023.

[19] P. Hager, F. Jungmann, R. Holland, et al. Evaluation and mitigation of the limitations of large language models in clinical decision-making. *Nature Medicine*, 30:2613–2622, 2024.

[20] N. F. Liu, K. Lin, J. Hewitt, et al. Lost in the middle: how language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[21] L. Zheng, W.-L. Chiang, Y. Sheng, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

[22] W. Kwon, Z. Li, S. Zhuang, et al. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[23] D. Sumanathilaka, L. Perera, and collaborators. Exploring the impact of temperature on large language models: a case study for classification task based on word sense disambiguation. In *Proceedings of the 7th International Conference on Artificial Intelligence and Machine Learning*, IEEE, 2025. `https://ieeexplore.ieee.org/document/11108362`

[24] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. GPTQ: accurate post-training quantization for generative pre-trained transformers. arXiv:2210.17323, 2022.

[25] Python Software Foundation. `asyncio` — Asynchronous I/O. Python documentation. `https://docs.python.org/3/library/asyncio.html`

[26] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC/IEEE 29119-1:2013 Software testing – Part 1: Concepts and definitions*. Geneva: ISO, 2013.

[27] K. Krekel, B. Oliveira, and A. Pfannschmidt. pytest: helps you write better programs. Python Package Index. `https://docs.pytest.org/en/stable/`

[28] Ned Batchelder and contributors. coverage.py: measuring code coverage in Python. `https://coverage.readthedocs.io/en/latest/`

[29] Model checking using Pydantic. https://pydantic.dev/

[30] S. Ramírez. FastAPI. `https://fastapi.tiangolo.com`

[31] SQLite Consortium. SQLite. `https://www.sqlite.org`

[32] Ollama. Ollama: run large language models locally. `https://github.com/ollama/ollama`

[33] G. Gerganov et al. llama.cpp: LLM inference in C/C++. `https://github.com/ggml-org/llama.cpp`

[34] A. Q. Jiang, A. Sablayrolles, A. Mensch, et al. Mistral 7B. arXiv:2310.06825, 2023.

[35] Mistral AI. Mistral Small 3. Model announcement, January 2025.

[36] A. Yang, A. Li, B. Yang, et al. Qwen3 technical report. arXiv:2505.09388, 2025.

[37] Gemma Team. Gemma 3 technical report. arXiv:2503.19786, 2025.

[38] A. Grattafiori, A. Dubey, A. Jauhri, et al. The Llama 3 herd of models. arXiv:2407.21783, 2024.

[39] JWT documentation. `https://www.jwt.io/introduction#what-is-json-web-token`.

[40] Why context length influences your bill. `https://c3x.dev/blog/gpu-memory-and-kv-cache-cost/`

[41] Papa Parse. `https://www.papaparse.com/docs`

A Terminology

EPR Electronic Patient Record: a digital record containing clinical information.

BI Business Intelligence department, the DZ department supporting data analysis.

DZ Deventer Hospital (*Deventer Ziekenhuis*).

LLM Large Language Model.

KV cache Key-value cache used by an autoregressive model to retain context during generation.

MoSCoW Prioritization method using Must, Should, and Could categories in this project.

B AI Statement

During the preparation of this work, the authors used ChatGPT and Claude to clarify language, support their understanding of relevant concepts, assist in developing arguments and assist in development. All the authors critically reviewed and edited all AI-generated content as necessary and take full responsibility for the final published work.

C Clinical system instruction

The LLM receives the following Dutch system instruction before the project-specific prompt. It provides the model with the intended role and general behavioral constraints.

Listing 1: Dutch clinical system instruction used by the worker

```
Je bent een klinisch expert en een zorgvuldige AI-assistent die onderzoekers
    ondersteunt bij het analyseren van Nederlandse klinische teksten. Je voert
    uitsluitend de gevraagde extractie- of classificatietaak uit op basis van de
    aangeleverde tekst.

Verzin geen informatie en gebruik geen medische kennis om ontbrekende gegevens in
    te vullen. Als informatie niet in de tekst staat of onvoldoende duidelijk is,
    gebruik dan de daarvoor toegestane null- of onbekende waarde volgens het
    opgegeven schema. Maak geen diagnose en geef geen behandeladvies. Behoud de
    betekenis van de bron, wees voorzichtig met ontkenningen, onzekerheid,
    tijdsaanduidingen en patintcontext, en behandel iedere rij afzonderlijk.

Volg de instructies van de gebruiker en het uitvoerschema nauwkeurig. Geef
    uitsluitend geldige JSON terug in het gevraagde formaat, zonder Markdown,
    toelichting of tekst buiten de JSON.
```

This instruction is deliberately conservative. The phrase “klinisch expert” defines the role and expected attention to clinical language; it does not mean that the model is a licensed healthcare professional or that its output is clinically validated. The model remains an analysis-support tool, and its responses must be validated against the output schema and reviewed by the researcher.

C.1 Machine-readable JSON instructions

The model-specific request fields are:

- For the OpenAI-compatible endpoint, the request contains `response_format: {"type": "json_object"}`. This asks the endpoint to return a JSON object. The project-specific prompt additionally states that the object must have the form `{"results": [{...}]}`.
- For Ollama, the request contains a `format` field with a JSON schema. The schema requires an object with a `results` property, whose value is an array of JSON objects. The detailed output-column schema is included in the project-specific prompt.

In simplified form, the relevant parts of the two request bodies are:

Listing 2: Format instructions in the LLM request bodies

```
# OpenAI-compatible endpoint
{
  "messages": [
    {"role": "system", "content": CLINICAL_SYSTEM_PROMPT},
    {"role": "user", "content": prompt}
  ],
  "response_format": {"type": "json_object"}
}

# Ollama endpoint
{
  "messages": [
    {"role": "system", "content": CLINICAL_SYSTEM_PROMPT},
```

```
      {"role": "user", "content": prompt}
    ],
    "format": {
      "type": "object",
      "properties": {
        "results": {"type": "array", "items": {"type": "object"}}
      },
      "required": ["results"]
    }
  }
```

D Algorithms

This appendix gives pseudocode for the three operations most relevant to correctness: dependency-aware scheduling, typed output validation with retry, and privacy-preserving resume. The pseudocode is language-independent; database and HTTP details are described in the design chapters.

D.1 Dependency-aware batch scheduling

Jobs become runnable only after their source jobs have completed. The round-robin queue prevents one project from monopolising the workers.

Listing 3: Dependency-aware scheduling and batch execution

```
while waiting_jobs is not empty or active_batches is not empty:
    for job in waiting_jobs:
        if all(source.status == COMPLETED for source in job.sources):
            initialise_runtime_state(job)
            move job from waiting_jobs to active_jobs
        else:
            rotate job to the back of waiting_jobs

    batch = next_batch(active_jobs, batch_size)
    if batch exists:
        prompt = construct_prompt(batch, job.template,
                                  job.columns, completed_source_outputs)
        response = call_llm(prompt, job.model_settings)
        valid_rows, invalid_rows = validate(response, job.output_schema)
        persist(valid_rows)
        requeue(invalid_rows)
```

D.2 Output validation and retry

Validation checks formatting and types, not semantic correctness. A well-formed but incorrect answer therefore remains a separate accuracy and human-review concern.

Listing 4: Typed output validation with bounded retry

```
function process_response(response, expected_ids, schema):
    rows = parse_json(response)
    invalid = []
    seen = set()

    for row in rows:
        if fields_or_types_are_invalid(row, schema):
            invalid.append(row)
        else if row.id not in expected_ids or row.id in seen:
            invalid.append(row)
        else:
            seen.add(row.id)
            persist(row)

    if invalid and retry_count < MAX_RETRIES:
        requeue(invalid)
    else if invalid:
        mark_job_and_project_failed()
```

D.3 Simple pause and resume

The source CSV is required again when a paused project is resumed. Only identifiers, typed outputs, and progress metadata are persisted and the CSV must be re-uploaded.

Listing 5: Resume without persisting source data

```
function pause(project):
    persist_completed_outputs(project)
    persist_status_and_progress(project)
    remove_runtime_csv(project)
    remove_project_from_queues(project)

function resume(project, uploaded_csv):
    validate_identifier_column(uploaded_csv, project.id_column)
    runtime_project = create_runtime_project(project, uploaded_csv)
    restore_completed_outputs(runtime_project)
    for job in runtime_project.jobs:
        rows = rows_without_persisted_output(job)
        if all_sources_completed(row, job):
            enqueue(rows)
    return project_status(runtime_project)
```

E Weekly sprint summary

Sprint boundaries are approximated by calendar week because the project chat does not mark them explicitly.

- **Pre-sprint (31 August–6 September): Project setup.** The Deventer Hospital project, a reusable on-premise LLM platform for healthcare, was assigned. GitLab was set up with epics, issues, one-week iterations, and labels for priority, type, and workflow, following an agile approach.
- **Sprint 1 (7–13 September): Context and scope.** The existing setup was identified as a Linux machine running Ollama with Mistral behind a Python API. It is disconnected from the network, so data arrives via USB. Logins and separate workspaces per user were identified as needed, together with a local database for jobs and results. Public or artificial datasets were sufficient for development, so an ethics-committee review was not required for this project.
- **Sprint 2 (14–20 September): Requirements.** The original codebase was analysed: CSV scripts, a FastAPI endpoint, Ollama and SQLite request logging, and no end-user interface. Requirements were set with MoSCoW priorities. Input is limited to CSV, with a selectable ID column and columns usable as prompt placeholders. Execution uses an asynchronous queue with priorities, progress reporting, and an error column. Only prompt templates and results are persisted, never source data. Conditional prompt sequences became a should-have, while auto-extending categories and multiple models became could-haves. A SQLite redesign and Linux setup-script redesign were out of scope. Constraints included one GPU, one concurrent user for now (up to four parallel workers), no hospital single sign-on, KV-cache limits, and rate limiting. A system diagram, report skeleton, mock-up, and first API code were started.
- **Sprint 3 (21–27 September): Architecture and first implementation.** A new API format and project structure were established. The frontend was built from the Figma design, including the prompt page and page switching. Backend jobs were changed to operate on pandas DataFrames and the first database changes were added. Registration replaced predefined users. One CSV per project was considered sufficient, with each prompt choosing its own columns. Prompt dependencies were added to the design backlog. The report structure was fixed around company and problem, requirements and constraints, design, results, testing, discussion, and appendices.
- **Sprint 4 (28 September–4 October): Frontend–backend integration.** The admin user was dropped from the MVP, and registration and login were wired up. Demonstration feedback led to simpler interface wording. Sample, range, and full execution replaced a proposed priority-style row selection. Source references, a knowledge graph, and a research question about processing speed were recorded as report ideas. Work moved to feature branches with frequent merges, and frontend work was split into connection, components, and styling. ID uniqueness checking, ten-row test runs, result viewing, and CSV download were added. Output validation remained a recommendation. CSV upload moved out of the prompt popup and the column selector became horizontally scrollable.
- **Sprint 5 (5–11 October): Output columns and project flow.** A refresh-token bug was fixed and token refresh was run on application load. Status polling was set to about five seconds and the default job state was corrected. Output columns were introduced with string, integer, number, boolean, and enum types. Results are returned as structured JSON or CSV. The CSV remains only in the frontend session and must be re-uploaded after login. Progress reporting, field renaming, and page-refresh routing were completed. Docker deployment, prompt testing, and result retrieval remained open. Model experiments and clearer task division were prioritized.
- **Sprint 6 (12–18 October): Job dependencies, filters, and large data.** Job dependencies were

implemented, including dependencies on unfinished jobs, and a “get all my jobs” endpoint was added. Projects, rather than individual jobs, can be run, paused, resumed, and cancelled. The separate test button was removed because the execution row range already provides that function. Columns were grouped by CSV source or previous job. Shared prompt templates were reinstated with a save/share checkbox. PapaParse addressed large CSV browser limits, with its worker option disabled for build-mode compatibility. Input filters, project controls, merged results, Ollama metadata collection, the Clini name, and a poster description were added or planned.

- **Sprint 7 (approximately week 8, 19–25 October): Deployment, testing, and documentation.** The system was deployed to the DZ and tested in Deventer using a roughly 500 MB CSV converted from a `.joblib` file. For the 24B Mistral model on the RTX 6000 Ada, the expected local generation range is approximately 40–60+ tokens/s, depending on quantization, engine, context, and batching. The deployment worked well overall and had only a minor browser permission issue. The templates page was merged and included sharing, tooltips, sample templates, and removal of unused files. Testing exposed issues with job-output labels, removing earlier CSV columns, editing input columns without re-uploading, and filters when two jobs shared an output column. Decisions included Dutch error messages, dimming unavailable buttons, confirmations for destructive actions, and nullable string outputs by default.
- **Sprint 8 (26-2 November): Finalization and Report planning.** The final sprint focused on stress testing and on revising the draft report. The report was restructured around a clearer storyline: the short introduction is merged into the domain analysis, "stakeholders" is used consistently, and a new rationale section explains how the constraints shaped the design decisions and what would have been done without them. Functional requirement 2.3 was removed because it is a design requirement rather than a user or system requirement. The choices for FastAPI, Pydantic and SQLite (versus Postgres) are to be justified explicitly.

F API Endpoints

Table 12 lists all endpoints of the production API. All endpoints except registration and login require an authenticated user. The `/debug` router is excluded because it is for development only (Section 3.3.3).

Method	Path	Description
Authentication		
POST	`/auth/register`	Register a user.
POST	`/auth/login`	Return an access token and set the refresh cookie.
POST	`/auth/refresh`	Issue a new access token from the refresh cookie.
GET	`/auth/users/me/`	Return the current username.
Projects and execution		
GET	`/project/all`	List the current user's projects.
POST	`/project/`	Create a project.
GET	`/project/{project_id}`	Get one project.
PATCH	`/project/{project_id}`	Update a project.
DELETE	`/project/{project_id}`	Delete a project.
GET	`/project/{project_id}/jobs`	List a project's jobs.
GET	`/project/{project_id}/results`	Get combined project results.
POST	`/project/run/{project_id}`	Upload CSV and queue execution.
GET	`/project/status`	Get project and job progress.
PATCH	`/project/pause/{project_id}`	Pause project processing.
PATCH	`/project/resume/{project_id}`	Resume with a CSV upload.
DELETE	`/project/cancel/{project_id}`	Cancel project processing.
Jobs		
GET	`/jobs/`	List the current user's jobs.
POST	`/jobs/`	Create a job.
GET	`/jobs/{job_id}`	Get one job.
PATCH	`/jobs/{job_id}`	Update a job.
DELETE	`/jobs/{job_id}`	Delete a job.
GET	`/jobs/results/{job_id}`	Get row-level results for a job.
Saved prompts		
GET	`/prompts/all`	List saved prompts.
POST	`/prompts/`	Create a saved prompt.
GET	`/prompts/{prompt_id}`	Get one saved prompt.
PATCH	`/prompts/{prompt_id}`	Update a saved prompt.
DELETE	`/prompts/{prompt_id}`	Delete a saved prompt.

Table 12: Endpoints of the production API.

G Client meeting notes

- **17 September 2026:** Discussed the client's workflow and requirements for analysing medical records. Researchers mainly work with CSV files containing free text and need to extract information into structured outputs such as scores, categories, or yes/no values. Noted the need to create, test, revise, and save prompts before running them on the full dataset. The interface should support row-based results, progress tracking, searching by identifiers, and datasets of up to 10,000 rows. The system must support asynchronous processing and prioritisation while preserving prompts and results for reproducibility. Source data should not be stored permanently.
- **23 September 2026:** Clarified that a project with multiple prompts should use a single CSV file, with each prompt selecting the columns it requires. Also noted the need to document dependencies between prompts in a flow diagram.
- **1 October 2026:** Discussed improvements to prompt testing and CSV handling. The user should be able to select specific rows, inspect the results, and adjust individual rows when debugging. Prompt testing should be limited to a maximum of 10 rows, while full-dataset results should be downloadable. Also noted the need to specify a unique identifier column and considered ways to validate LLM outputs, such as manually checking a sample or using another LLM as a judge. Included output validation as a recommendation in the final report and investigate possible alternatives to Mistral.
- **15 October 2026:** Reviewed the prototype through a demonstration and identified usability improvements, including short explanations of interface functionality. Also noted that the report should address ethical and legal responsibility for LLM-generated results. Output validation and the handling of large CSV files remained important topics to investigate.

H Supervisor meeting notes

- **9 September 2026** Discussed the existing system: a Linux server running Ollama and Mistral, accessed through a Python API. Requirements included CSV-based processing, a local database for jobs and results, prompt testing on individual rows, and secure, user-specific access. Public healthcare datasets were considered to address ethical and data-access concerns.
- **16 September 2026** Discussed hardware constraints and model deployment. Determine the available GPU hardware (NVIDIA Ada 6000), GPU memory requirements, context window limits, and concurrent sequences per user. Investigate asynchronous processing for parallel jobs and whether the existing SQLite database requires redesign.
- **23 September 2026** Discussed the final report structure: company and problem introduction, requirements and constraints, high- and low-level design, system architecture, GUI, models, testing, results, discussion, and user/developer guides. Document model context windows and inference performance, including inter-token latency. The client also requested a flow diagram showing prompt dependencies and clarified that prompts within a project share one CSV dataset.
- **30 September 2026** Discussed implementation about agentic AI, must confirm with client how this implementation could work out eventually.
- **7 October 2026** The current prototype is there, but the team must experiment with different LLMs and models to increase technical depth and improve the project's quality.
- **14 October 2026** Reviewed outstanding functionality and priorities: project/job management, filtering, pause/resume/cancel, prompt templates, CSV column selection, and job dependencies. Investigate Ollama performance metrics, including token counts, inter-token latency, and generation speed; test multiple jobs and report the results. Clarify prompt-testing and row-range behaviour, simplify redundant UI controls, and improve usability based on user feedback.
- **28 October 2026** Reviewed report feedback and identified required improvements: restructure the introduction, improve the storyline, use consistent stakeholder terminology, and convert section 2.5 into a flowchart. Add justification for design decisions and constraints, explain technology choices (FastAPI, Pydantic, SQLite/PostgreSQL), verify the 8K token storage calculation, align sequence and class diagrams, move diagrams into the main report, and add a discussion and reflection chapter before the conclusion.

I User test plan

I.0.1 Instructie voor de deelnemer

Tijdens deze test vragen we je om een aantal opdrachten uit te voeren in Clini. Probeer de opdrachten zelfstandig uit te voeren. Er zijn geen goede of foute antwoorden; we testen de interface, niet jou.

Het doel van dit onderzoek is om het systeem “Clini” te testen op gebruiksvriendelijkheid. Tijdens de sessie (duur: ongeveer 10 minuten) vragen we je om opdrachten uit te voeren in het systeem en hier vragen over te beantwoorden. Je deelname is volledig vrijwillig. Je kunt op elk moment stoppen met het onderzoek, zonder opgaaf van reden. Jouw gegevens worden dan niet gebruikt in onze resultaten.

Gebruik uitsluitend de testgegevens die door ons beschikbaar zijn gesteld.

I.0.2 Deel 1 – Taken

1. Maak een account aan en log in op Clini.
2. Open het taakoverzicht, zoek een bestaande taak en bekijk de status.
3. Maak een nieuw project aan.
4. Maak binnen het project een prompt aan, configureer de CSV-kolommen en het uitvoertype, en sla de prompt op.
5. Upload het testbestand, selecteer de unieke identificatiekolom, test de prompt op maximaal 10 rijen en bekijk de testresultaten.
6. Start een analyse, bekijk de voortgang en pauzeer en hervat de analyse indien beschikbaar.
7. Bekijk de resultaten per rij, zoek op identificatie en exporteer de resultaten.
8. Sla een prompt op als sjabloon en controleer of deze opnieuw kan worden gebruikt.

I.0.3 Deel 2 – Vragen achteraf

Algemene ervaring

1. Hoe vond je het om met Clini te werken?
2. Was het duidelijk waar je moest beginnen?
3. Kon je gemakkelijk vinden wat je zocht?
4. Was de interface overzichtelijk?

Projecten en prompts

5. Was het eenvoudig om een project aan te maken?
6. Was het duidelijk hoe je een prompt moest instellen?
7. Waren de invoer- en uitvoeropties begrijpelijk?

Bestanden, analyses en resultaten

8. Was het uploaden van het CSV-bestand duidelijk?
9. Was het duidelijk hoe je een prompt kon testen?
10. Kon je de voortgang van de analyse goed volgen?
11. Waren de resultaten begrijpelijk weergegeven?
12. Was het eenvoudig om de resultaten te exporteren?

Verbeterpunten

13. Bij welke taak liep je het meest vast?
14. Waren er knoppen, teksten of functies die onduidelijk waren?
15. Ontbrak er volgens jou een functie?
16. Wat zou je als eerste aan Clini verbeteren?

Afsluiting

1. Is er nog iets dat je wilt toevoegen?
2. Zou je Clini in je werk willen gebruiken? Waarom wel of niet?

J User test answers

J.0.1 Algemene ervaring

Vraag 1: Hoe vond je het om met Clini te werken?

Deelnemer 1: Prettig programma om mee te werken. Redelijk simplistisch gebouwd en daardoor gebruiksvriendelijk.

Deelnemer 2: Ik vind het een overzichtelijk, gebruiksvriendelijk programma dat het werken met prompts enorm vergemakkelijkt.

Deelnemer 3: Prima, werkt goed, snel en intuïtief.

Vraag 2: Was het duidelijk waar je moest beginnen?

Deelnemer 1: Na even zoeken wel. In de meegeleverde taken stond het verkeerd, maar daarna was het goed te doen.

Deelnemer 2: De stappen in de handleiding staan wat door elkaar, aangezien er eerst een project en prompt moet worden aangemaakt voordat je de status van een taak kan bekijken.

Deelnemer 3: Ervan uitgaande dat het begin het moment van inloggen is, dan is het duidelijk waar je moet beginnen.

Vraag 3: Kon je gemakkelijk vinden wat je zocht?

Deelnemer 1: Over het algemeen wel. Ik moest wel eerst even zoeken hoe ik een mislukte taak opnieuw kon starten.

Deelnemer 2: Over het algemeen wel, de lay-out is overzichtelijk.

Deelnemer 3: Ja.

Vraag 4: Was de interface overzichtelijk?

Deelnemer 1: Binnen één overzicht zijn veel aspecten overzichtelijk geplaatst, dat is erg prettig.

Deelnemer 2: Ja.

Deelnemer 3: Ja.

J.0.2 Projecten en prompts

Vraag 5: Was het eenvoudig om een project aan te maken?

Deelnemer 1: Dat was zeker eenvoudig.

Deelnemer 2: Ja.

Deelnemer 3: Ja.

Vraag 6: Was het duidelijk hoe je een prompt moest instellen?

Deelnemer 1: Bij prompt aanmaken is het even goed kijken, maar alles is wel logisch.

Deelnemer 2: Na een korte uitleg van een collega was dit snel duidelijk.

Deelnemer 3: Ja.

Vraag 7: Waren de invoer- en uitvoeropties begrijpelijk?

Deelnemer 1: Het uploaden van de CSV is vanzelfsprekend. Wel is het even uitvogelen dat je eerst de CSV moet uploaden.

Deelnemer 2: Het is prettig om een korte introductie van een collega te krijgen; over het algemeen is het begrijpelijk.

Deelnemer 3: Ja.

J.0.3 Bestanden, analyses en resultaten

Vraag 8: Was het uploaden van het CSV-bestand duidelijk?

Deelnemer 1: Het uploaden zelf was duidelijk en gemakkelijk.

Deelnemer 2: Ja.

Deelnemer 3: Ja.

Vraag 9: Was het duidelijk hoe je een prompt kon testen?

Deelnemer 1: Door een beperkte hoeveelheid rijen te selecteren is het testen goed te doen.

Deelnemer 2: Ja.

Deelnemer 3: Ja.

Vraag 10: Kon je de voortgang van de analyse goed volgen?

Deelnemer 1: De voortgang is prima te volgen, maar kleinere stappen dan per 50 rijen zouden mogelijk duidelijker zijn.

Deelnemer 2: Ja.

Deelnemer 3: De voortgangsbalk bleef 90% van de tijd op 0% om vervolgens naar 50% en 100% te gaan.

Vraag 11: Waren de resultaten begrijpelijk weergegeven?

Deelnemer 1: Middels het oogje is het fijn dat je de resultaten in kan zien en kan checken.

Deelnemer 2: Ja, ook tijdens het runnen werkt het oogje mooi.

Deelnemer 3: Ja.

Vraag 12: Was het eenvoudig om de resultaten te exporteren?

Deelnemer 1: De downloadknop was goed te vinden.

Deelnemer 2: Ja.

Deelnemer 3: Ja.

J.0.4 Verbeterpunten

Vraag 13: Bij welke taak liep je het meest vast?

Deelnemer 1: Ik wist bij de eerste run niet goed of de run gestart was, omdat het enkele minuten duurde voordat de eerste 50 rijen klaar waren.

Deelnemer 2: Vooral de optie om output van geslaagde prompts als input te gebruiken behoeft extra uitleg.

Deelnemer 3: Ik liep niet vast.

Vraag 14: Waren er knoppen, teksten of functies die onduidelijk waren?

Deelnemer 1: -

Deelnemer 2: Zie hierboven.

Deelnemer 3: Nee.

Vraag 15: Ontbrak er volgens jou een functie?

Deelnemer 1: Het kunnen meenemen van kolommen uit de input-CSV in de output-CSV zou controle van de LLM-resultaten makkelijker maken.

Deelnemer 2: Werkbladkeuze voor Excel, pauzeren binnen een taak en foutmeldingen bij mislukte taken zouden nuttig zijn.

Deelnemer 3: Nee.

Vraag 16: Wat zou je als eerste aan Clini verbeteren?

Deelnemer 1: De optie om kolommen te selecteren die je mee wilt nemen in de output.

Deelnemer 2: Ik vind het een mooie basis om op verder te bouwen en zou bovenstaande punten toevoegen.

Deelnemer 3: De voortgangsmonitoring: dynamische rapportage zou duidelijker kunnen zijn.

J.0.5 Afsluiting

Vraag A1: Is er nog iets dat je wilt toevoegen?

Deelnemer 1: Hele mooie app gemaakt jongens, goed werk geleverd. Dik tevreden.

Deelnemer 2: Zie mijn opmerkingen onder vraag 15.

Deelnemer 3: Zie antwoord op vraag 16.

Vraag A2: Zou je Clini in je werk willen gebruiken? Waarom wel of niet?

Deelnemer 1: Ik ben van mening dat dit een verbetering is ten opzichte van de eerdere Python-scripts.

Deelnemer 2: Ja, het is een gebruiksvriendelijke tool die goed gebruik maakt van het LLM.

Deelnemer 3: Ja, omdat het gebruik van het LLM eenvoudiger en overzichtelijker wordt.

K Risk assessment (P×I)

The risks of Table 5 were scored with the PXI method. Probability (P) and impact (I) are each rated from 0 to 5, where 0 means none and 5 means very high (P: almost certain to occur; I: blocks a requirement, the deployment or the validity of the results). The risk score is $P \times I$, with a maximum of 25. Scores of 0–6 are rated low, 8–12 medium and 15–25 high. The scores take into account the likelihood, impact and observed outcome in Table 5.

ID	Risk	P	I	P×I	Rating
R1	No approved real patient data for development	5	4	20	High
R2	Isolated hospital machine prevents remote testing	5	4	20	High
R3	GPU memory exhaustion or CPU spillover	3	4	12	Medium
R4	Hallucinated or malformed LLM output	4	5	20	High
R5	Deployment failure (dependencies, configuration)	3	4	12	Medium
R6	Scope growth during a short project	3	3	9	Medium

Table 13: P×I risk assessment.

	I = 1	I = 2	I = 3	I = 4	I = 5
P = 5				R1, R2	
P = 4					R4
P = 3			R6	R3, R5	
P = 2					
P = 1					

Table 14: Risk matrix (probability and impact)

R1, R2 and R4 are rated high. R1 and R2 are related to the project environment, and R4 is inherent to LLMs. This is the reason for the conditions in Section 6.