UNIVERSITAT D'ALACANT
UNIVERSIDAD DE ALICANTE
Facultat de Ciències Econòmiques i Empresarials
Facultad de Ciencias Económicas y Empresariales

MÁSTER UNIVERSITARIO EN ECONOMÍA CON CIENCIA DE DATOS

CURSO ACADÉMICO 2025 – 2026

REVIN: An Automated Pipeline for the Regulatory Review of Real-Estate Appraisal Reports with Large Language Models

The REVIN project (REvisión de Valoraciones INmobiliarias)

Jorge Pérez Amat

Ignacio Ribelles Ferrándiz

Instituto de Análisis Inmobiliario, S.L.U. (Instai), Grupo Euroval

San Vicente del Raspeig, 20 de septiembre de 2026

MÁSTER UNIVERSITARIO EN ECONOMÍA CON CIENCIA DE DATOS

Curso académico 2025 – 2026

REVIN: An Automated Pipeline for the Regulatory Review of Real-Estate Appraisal Reports with Large Language Models

The REVIN project (REvisión de Valoraciones INmobiliarias)

Estudiante:	Jorge Pérez Amat
Tutor de empresa:	Ignacio Ribelles Ferrándiz
Empresa:	Instituto de Análisis Inmobiliario, S.L.U. (Instai), Grupo Euroval
Programa:	Máster Universitario en Economía con Ciencia de Datos Facultad de Ciencias Económicas y Empresariales, Universidad de Alicante

San Vicente del Raspeig, 20 de septiembre de 2026

Abstract

This thesis describes REVIN, a software system that automatically reviews real-estate appraisal reports. In Spain, before granting a mortgage, banks require an official appraisal, which must follow Order ECO/805/2003. These reports are reviewed by hand, which is slow and easy to get wrong. REVIN was designed and built during a curricular internship in a private company.

The system is a pipeline of four independent blocks. The first receives the PDF and identifies it by the hash of its content. The second extracts the text and the tables. The third uses a large language model to turn the report into structured data in JSON format, where every value keeps a literal quotation from the document as evidence. The fourth applies the business rules as deterministic checks in code and produces a result that can be audited. The company set one principle at the start, and the whole design follows from it: the model only extracts data and the code applies the rules, so the model takes no decisions. There is one prompt per appraisal company, and only one company was studied during the internship.

This work is descriptive. It documents the design, the architecture, the development process, including the decisions that were later changed, and the problems found during the work. The thesis reports no measured results. The source code and the corpus of reports stay with the company, so no reader could check a figure given here. Instead, it sets out the protocol that such a measurement should follow, at three levels of cost, together with the threats to its validity. Confidentiality is also why the thesis uses pseudocode instead of real code, and why the companies in the sample documents are not named.

Keywords: large language models, information extraction, document processing, regulatory compliance, real-estate valuation.

Resumen

Este Trabajo de Fin de Máster describe REVIN, un sistema informático que revisa de forma automática informes de tasación inmobiliaria. En España, los bancos exigen una tasación oficial, que debe cumplir la Orden ECO/805/2003, antes de conceder una hipoteca. Estos informes se revisan a mano, una tarea lenta y propensa a errores. REVIN se diseñó y se construyó durante unas prácticas curriculares en una empresa privada.

El sistema es un *pipeline* de cuatro bloques independientes. El primero recibe el PDF y lo identifica mediante el *hash* de su contenido. El segundo extrae el texto y las tablas. El tercero usa un modelo de lenguaje grande para convertir el informe en datos estructurados en formato JSON, donde cada valor conserva una cita literal del documento como evidencia. El cuarto aplica las reglas de negocio como comprobaciones deterministas en código y produce un resultado que se puede auditar. La empresa fijó un principio desde el inicio, y de él sale todo el diseño: el modelo solo extrae datos y las reglas las aplica el código, de modo que ninguna decisión la toma el modelo. Hay un *prompt* por sociedad de tasación, y durante las prácticas se estudió una sola.

Este trabajo es descriptivo. Documenta el diseño, la arquitectura, el proceso de desarrollo, incluidas las decisiones que se cambiaron después, y los problemas encontrados durante el trabajo. No presenta resultados medidos. El código fuente y el corpus de informes permanecen en la empresa, así que nadie que lea esto podría comprobar una cifra. En su lugar, especifica el protocolo que debería seguir esa medición, en tres niveles de coste, junto con las amenazas a su validez. La confidencialidad es también la razón de que el trabajo use pseudocódigo en lugar de código real y de que no se nombre a las empresas de los documentos de muestra.

Palabras clave: modelos de lenguaje grandes, extracción de información, procesamiento de documentos, cumplimiento normativo, valoración inmobiliaria.

Confidentiality Statement

This thesis describes work carried out during a curricular internship at Instituto de Análisis Inmobiliario, S.L.U. (Instai), part of the Euroval group. The source code of the REVIN project, the instructions given to the language model (the system prompts), and the business data used during development are the property of the company and are covered by a confidentiality agreement.

For this reason, this document does not reproduce source code, the production prompts, or the spreadsheet of controls that the company provided as the source of the validation rules. The technical components are described through pseudocode and high-level explanations, and the catalogue of checks is described by category rather than rule by rule. No fragment of any real appraisal report is reproduced. The companies that appear in the sample documents used during development, namely the appraisal company that issued the reports and the financial institution that requested them, are not named and are referred to in general terms.

The descriptions in this thesis are written so that the system and its design can be understood and evaluated without disclosing material that belongs to the company.

Contents

List of Figures

List of Tables

List of Pseudocodes

Acronyms

Acronym	Meaning
AI	Artificial Intelligence
API	Application Programming Interface
AVM	Automated Valuation Model
ECO/805/2003	Order ECO/805/2003, the Spanish ministerial order on the valuation of property for financial purposes
IDUFIR	*Identificador Único de Finca Registral*, the unique identifier of a registered property
JSON	JavaScript Object Notation
LLM	Large Language Model
LTV	Loan-to-Value ratio
OCR	Optical Character Recognition
OOP	Object-Oriented Programming
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation
RegTech	Regulatory Technology
SHA256	Secure Hash Algorithm, 256-bit variant
TFM	*Trabajo de Fin de Máster* (Master's Thesis)
VPO	*Vivienda de Protección Oficial* (publicly protected housing)

Chapter 1
Introduction

In one of the reports I processed during this project, the surface area of the dwelling stated in the appraisal did not match the one recorded in the land registry note. Neither figure was invented: the two documents were most probably measuring different things, since a property has several surfaces and they are not interchangeable. But the report gave one number and the registry gave another, without saying why, and a report that reaches a bank in that state is defective.

Finding that kind of disagreement is what a review of an appraisal report is for, and until now it has been done by reading the documents side by side. This thesis describes REVIN (*REvisión de Valoraciones INmobiliarias*), a software system that performs that review automatically. The system was designed and built during a curricular internship in a private company. This chapter explains the problem that motivated the work, the setting in which it was carried out, the objectives, and the limits of what this document can show.

1.1 Context and motivation

In Spain, a bank that grants a mortgage needs an official appraisal of the property used as collateral. These appraisals are produced by appraisal companies that the Bank of Spain approves, registers and supervises, and they must follow Order ECO/805/2003 (Ministerio de Economía, 2003) together with the related regulation, such as Royal Decree 716/2009 (Ministerio de Economía y Hacienda, 2009) and Circular 4/2017 of the Bank of Spain (Banco de España, 2017).

An appraisal report is a long PDF, commonly around sixty pages and sometimes well over a hundred, that combines the appraiser's own text with documents from other sources: the land registry note, the cadastral record, photographs and plans. Before the report reaches the bank it is reviewed. The reviewer checks, among other things, that the surface areas agree between

the report, the registry note and the cadastre, that the relevant dates have not expired, that the required annexes are present, and that the owners listed are the same in every document.

The purpose of that review is precisely to prevent a report from reaching the lender with defects such as the one described above. The review is a control internal to the valuation process, carried out before the document leaves for the bank.

Automating the review is worth doing because of the nature of the task. The review is systematic, repetitive, and consists almost entirely of comparing values that appear in several places within the same document. Those same features make it vulnerable to human error. A reviewer comparing dozens of figures across a report of that length, one report after another, will eventually read the right number in the wrong place, and a report with a defect that slipped through looks exactly like a report with none.

REVIN automates this review. The system receives the appraisal PDF, extracts its content, and applies a set of checks that produce a result a person can audit.

1.1.1 Why this problem fits this master's degree

The problem sits where the two sides of the degree meet. The domain is economic, because the value fixed by an appraisal feeds decisions of the lender that Section 2.1 describes in detail. The solution is applied data science, because it extracts information from unstructured documents with a large language model and then applies fixed rules in code.

The part of the degree that served me most directly was programming, although what it taught was the concept of a class in Python rather than how to build a system with objects. I had to learn much of what this project required on my own while building it, including object-oriented design and how a language model is prompted, and Chapter 8 returns to that.

1.2 The host company and the internship

The work presented in this thesis was carried out during a curricular internship at Instituto de Análisis Inmobiliario, S.L.U. (Instai), a company in the Euroval group that works on property valuation and on data tools for that sector. The group was founded in Alicante in 1990 and has been approved by the Bank of Spain since that same year, and it supports a network of appraisers working across Spain and Portugal. The review that this project automates is therefore carried out in Alicante, on reports that arrive from all of that network. My company tutor was Ignacio Ribelles Ferrándiz, head of the data area.

1.2.1 How the project reached me

REVIN was not invented as an internship project. The head of the company had been asking for it over a long period, and it had been sketched in 2025 without going any further. When I arrived, my tutor saw the opportunity to take it up again and to hand it to me as the subject of both my internship and this thesis. We began by going through the ideas that had been considered earlier, and built from there.

Nothing had been implemented before. The project started from an empty folder, which is the reason this document can describe every decision from its origin.

The way I was supervised shaped the work. My tutor is an experienced professional who had automated other processes in the company, and he could have handed me a design to implement. He did the opposite. He gave me freedom to try, let me get stuck, and stepped in when I had been stuck long enough, or when I showed him something I believed was finished, to tell me what was wrong with it. Several of the changes reported in Chapter 7 are the result of that method, and so is most of what I learned.

The project also had weekly meetings with the rest of the team, the people who would eventually use or maintain the tool. Those meetings were where the checks were clarified, one by one, and where I was told which additional ones would be needed. Two ideas about the future of the system also came out of those meetings: running the language model on the company's own hardware, and adding optical character recognition.

1.2.2 My role

The data team is small, so each member takes on more than one role. In this project I worked as a data analyst, a data scientist and a data engineer at different moments, and I was responsible for building the REVIN pipeline and for its technical direction, which includes the decisions reported in Chapter 7 and the choices about what to discard. Some of the decisions behind the system came from the company or from my tutor rather than from me, and Section 1.4 separates the two. The code itself was written with the assistance of the tools described in Section 3.7.

1.3 First contact with the documents

The length of the reports was not what made the problem difficult. What made it difficult was that every appraisal company writes them differently.

The sample I was given contained reports from several companies, and they did not agree on where anything was. The same value lived in a different section, under a different heading, in a different table, depending on who had produced the document. Some reports had no selectable text at all, because they had been scanned, and would need optical character recognition before anything could be read from them. In others the tables were broken in ways that made the information hard to recover, or survived the conversion to text badly.

I did not know how to approach that, and the answer came from my tutor: not all at once, but one appraisal company at a time, studying where that company puts each piece of information and then writing instructions specific to it. We chose one of the largest companies, whose reports are of good quality, as the starting point, and a considerable part of the internship went into reading its documents closely enough to describe their structure precisely. That decision explains much of the design that follows, and Chapter 7 reports how it was first resisted and then adopted.

1.4 Problem statement and objectives

The goal of the project was to replace the manual review of appraisal reports with an automatic one: the system has to read an appraisal PDF, organise its content, and check that content against the rules a human reviewer would apply. The general objective of my work was to design and build REVIN as a working, pre-production pipeline.

The company asked for some of these things and I decided others, and the difference matters.

1.4.1 Objectives set by the company

- **A modular architecture of independent blocks**, ready to be deployed on Microsoft Azure. My tutor asked for this explicitly. My first version was a single program, and restructuring it into separate blocks was one of the larger changes of the project.
- **Structured extraction with a language model.** The company wanted the model to turn the report into structured data, in JSON or in any equivalent format, so that the next step could work with it easily.
- **Validation in code, never by the model.** This was stated from the start, with two reasons given: calling the model again to validate would increase the cost, and, more

importantly, a judgement made by a model cannot be checked as easily as one made by a rule. This requirement turned out to be the principle the whole system rests on.

1.4.2 Decisions I contributed

- **How the PDF becomes text.** Here I was given complete freedom. I investigated the available Python libraries and first produced a plain-text version. My tutor then suggested that Markdown might suit a language model better, and comparing the two showed he was right, because Markdown preserved the structure of the tables. I also added a requirement of my own: to convert the document page by page rather than all at once, so that each page could be marked with its number.

- **Evidence for every value.** The page marking described above is what made it possible to impose a stricter requirement: every value the model extracts must be accompanied by the literal fragment of the report it was taken from, together with the page on which that fragment appears. A value that cannot be quoted is left empty. This requirement did not form part of the original specification, and Chapter 2 situates it with respect to the literature on attribution.

- **A contract of states between the blocks.** Each block ends by reporting *what happened* to the document rather than merely whether it succeeded, which allows the pipeline to distinguish a genuine malfunction from a document that is correct but cannot be processed. Section 3.3 develops this distinction.

- **A single definition of the data schema.** The structure of the extracted data is described in one place only, so that adding or modifying a field does not require the same change to be made twice.

1.4.3 What was not achieved

The base of the system was delivered within the period of the internship, but several objectives remained open. They were extending the system to further appraisal companies, completing the catalogue of checks, deploying the pipeline on Azure, finishing the preparation for local language models, and establishing the reliability of the extraction through measurement. Chapter 9 returns to them in detail. The last one is the most consequential, and Chapter 6 is devoted to how it should be carried out.

1.5 Scope and confidentiality constraints

This thesis is descriptive. It documents the design of the system, its architecture, the development process, including the decisions that were later changed, and the problems found during the work. It does not report measured results.

The reason is practical. When the internship ended, I lost access to the code and to the corpus of appraisal reports, both of which are the property of the company, and with that access went the possibility of studying the system's accuracy. Any figure reported here would therefore be one that neither the reader nor I could reproduce. In its place, Chapter 6 gives the protocol such a study should follow.

The code and the model instructions, the system prompts, are the property of Instai and are covered by a confidentiality agreement. For this reason the thesis does not reproduce real code or the prompts. Instead, it explains the components with pseudocode and with descriptions at a conceptual level. The companies that appear in the sample documents used during development, the appraisal company that issued the reports and the bank that requested them, are not named, and are referred to in general terms.

1.6 Structure of this document

The rest of the thesis is organised as follows. Chapter 2 reviews related work on information extraction with language models, on grounding and the control of hallucinations, on document processing, and on the automation of regulatory compliance. Chapter 3 gives an overview of the solution and the working method. Chapter 4 describes how the project was planned and managed. Chapter 5 explains the implementation block by block, with the flow diagram of each one, and the corresponding pseudocodes are collected in Appendix F. Chapter 6 sets out the protocol by which the system should be evaluated. Chapter 7 discusses the main design decisions and the lessons learned. Chapter 8 presents the conclusions, and Chapter 9 describes the future work. The appendices collect supporting material.

Chapter 2
State of the Art and Background

This chapter reviews the work that this thesis builds on. It covers five lines, each of which touches a different part of the system. They are the domain and its regulation, the use of language models to extract structured information from documents, the control of hallucination through grounding and attribution, the processing of documents (that is, how a PDF becomes text a model can read), and the automation of regulatory compliance.

2.1 Domain and regulatory background

The system built in this project reviews appraisal reports. That review matters because of the chain of events in which an appraisal report appears.

A person asks a bank for a mortgage to buy a property. The bank will lend the money with that property as the guarantee: if the loan is not repaid, the bank can recover the property and sell it. The bank therefore needs to know what the property is worth, and it cannot simply take the buyer's or the seller's word for it. This is why an independent valuation is required.

In Spain that valuation is not a free-form opinion. It has to be produced by an appraisal company that the Bank of Spain has approved, registered and supervises, and it has to follow Order ECO/805/2003 (Ministerio de Economía, 2003), which sets out the methods that may be used to reach a value, the checks the appraiser must carry out, and what the resulting report must contain. Two further rules complete the picture. Royal Decree 716/2009 (Ministerio de Economía y Hacienda, 2009) states the conditions a valuation must meet for the loan to be eligible as backing for mortgage bonds, which are the securities banks issue to finance their lending. Circular 4/2017 of the Bank of Spain (Banco de España, 2017) sets the accounting rules banks apply to their loans, including how the value of the guarantee affects the provisions the bank must set aside against the risk of not being repaid.

2.1.1 Why an error in the report is not only a documentary problem

The number written in an appraisal report does not stay inside the report. It determines the loan-to-value ratio, that is, how much money is lent against how much the property is worth. It determines whether the loan can be used as backing for mortgage bonds. And it feeds the calculation of how much capital the bank must hold against that loan.

An appraisal that is wrong, or that does not comply with the order, therefore has effects on the bank's balance sheet, and, when many such appraisals are wrong in the same direction, on how accurately risk is measured in the mortgage market as a whole. That is why these reports are reviewed carefully, and why automating the review was worth the effort.

2.1.2 Valuing a property and reviewing a valuation are different problems

There is an established body of work on automating the valuation itself. Automated valuation models estimate the price of a property from its characteristics and from market data, historically with statistical regressions and, more recently, with machine learning methods. Stang et al. (Stang, Krämer, Nagl, & Schäfers, 2022) review this evolution and what it means in practice for the appraisal profession. This literature asks one question: *what is this property worth?*

REVIN does not ask that question, and the two are easily confused. REVIN takes an appraisal that a certified professional has already produced, and asks whether the document is complete, internally consistent and compliant with the order: whether the surface areas agree across the report, the land registry note and the cadastral record, whether the required annexes are present, whether the dates are still valid. It never produces a value of its own, and it never contradicts the appraiser's judgement.

The two lines of work are complementary. One estimates value, the other checks that a valuation was carried out properly. The problem addressed in this thesis belongs to the second, and it has received considerably less attention.

2.2 Information extraction from documents with language models

Information extraction turns free text into structured data. For many years this was done with hand-written rules, or with models trained specifically for each field. Both approaches

require an effort proportional to the number of fields and to the variety of documents, which is why they scale badly when the documents come in many different formats.

Large language models changed this. Brown et al. (Brown, Mann, Ryder, et al., 2020) showed that a sufficiently large model can perform a task it was never trained on, simply from a few examples placed in the prompt. This removed the need for annotated training data for every new field. Later work builds on that idea and uses language models to extract entities, relations and events directly, as reviewed in the survey by Xu et al. (D. Xu et al., 2024).

2.2.1 From examples to instructions

The situation has moved on since 2020 in a way that matters for this project. The result of Brown et al. concerned examples: show the model a handful of solved cases and it generalises from them. Current models are also trained to follow instructions, and in practice a detailed, well-organised set of written rules is often enough on its own, with no examples at all.

REVIN relies on that. Its prompt contains no worked examples. It contains a description of where each value is located in that appraisal company's report, the shape the answer must take, and a short list of rules the model must respect, such as copying the text literally and never calculating anything. That matters for a system that pays per use. Examples occupy space in the prompt, and every document sent to the model already occupies a great many pages, so a prompt made of rules rather than examples leaves more room for the document itself.

This approach fits appraisal reports well. The documents are long, their structure varies between appraisal companies, and the number of values to extract is in the order of a hundred. Writing and maintaining a separate rule in code for every field, for every company, would be costly and fragile. REVIN therefore asks a language model to read the report and return its data as a JSON object.

2.2.2 Getting a machine-readable answer

Asking a model for structured data raises a practical problem: the answer must be valid JSON every time, or the next step of the pipeline has nothing to work with. Geng et al. (Geng, Josifoski, Peyrard, & West, 2023) address this with grammar-constrained decoding, a technique that restricts what the model is allowed to generate so that the output is guaranteed to follow a given formal structure, and they show that constrained models do better than unconstrained ones on structured tasks. Commercial providers now offer a simpler version of

the same guarantee: a mode in which the model is required to answer with a JSON object.

REVIN uses that provider-level guarantee rather than defining a grammar of its own, together with a randomness setting of zero so that the same document yields the same answer. The choice is deliberately modest. The shape of the data is described in the prompt, in ordinary language, and the format is enforced by the API. Defining a formal grammar would give a stronger guarantee, but it would have to be maintained alongside the prompt, and keeping the same information in two places is precisely the duplication that Chapter 7 reports having removed.

2.3 Grounding, attribution and hallucination control

The main risk of language models is hallucination, that is, producing text that reads well but is not supported by the source. Ji et al. (Ji et al., 2023) survey this problem and the methods proposed to reduce it. One common method is retrieval-augmented generation, introduced by Lewis et al. (Lewis et al., 2020), which gives the model passages retrieved from a collection of documents so that its answer is based on real text rather than on what it remembers.

A related line of work goes further and asks that the grounding itself be *verifiable*, so that a reader can check where an answer came from. Rashkin et al. (Rashkin et al., 2023) propose a framework in which a statement counts as attributable only if an identified source supports it, and they formalise how that property can be evaluated. Gao et al. (Gao, Yen, Yu, & Chen, 2023) take the step from evaluating to generating: they make language models produce answers that carry citations to the passages supporting them, and introduce a benchmark for measuring the quality of those citations. Their result is a warning for any system that relies on this idea, since even the best models they tested failed to fully support their statements a large part of the time.

In a document such as an appraisal report the situation is more favourable than in those studies, and REVIN takes advantage of the difference. The source is not a collection of documents to search through. It is the single document being processed, and it is present in full in the prompt. The requirement can therefore be made stricter than a citation to a passage: the model must copy the exact text, and a value it cannot copy is left empty. Section 3.4 states the rule in full.

Because the quotation has to be an exact substring of the document, a program can decide whether the attribution holds, with no human and no second model needed to judge it. That is what places this work with respect to the literature above.

2.3.1 Long documents and the limits of the context window

An appraisal report is long. Most of those used in this project ran to around sixty pages, but the sample also contained documents of eighty, and some exceeding a hundred. That places them in the range where the way a model handles long inputs starts to matter.

Liu et al. (Liu et al., 2024) show that language models do not use long inputs evenly. Performance depends on *where* in the input the relevant information sits, and it is noticeably worse when that information lies in the middle rather than near the beginning or the end. Since the values REVIN extracts are spread throughout the report, this is a limitation the design has to live with. It is one reason why the prompt states explicitly where each section of the document is expected to be found, instead of leaving the model to search for it.

2.4 Document processing: from PDF to text

Before a model can read a report, the document has to be turned into text. For PDFs that contain real text, software libraries can extract the characters and the tables directly. For pages that are images, optical character recognition is needed, and the Tesseract engine (Smith, 2007) is the best-known open-source example. A different line of work models the text together with its position on the page: LayoutLM, by Xu et al. (Y. Xu et al., 2020), trains a model on text and layout jointly for tasks such as understanding forms and receipts.

This step is often treated as a solved preliminary, and it is not. Recent work has made that visible by measuring it. READoc (Li et al., 2024) treats the conversion of a PDF into structured Markdown as a task in its own right and provides a benchmark for it, and OmniDocBench (Ouyang et al., 2025) evaluates document parsing across document types, layouts and languages, with separate scores for text, tables, formulas and reading order. Open toolkits have appeared alongside these benchmarks, such as Docling (Livathinos et al., 2025), which converts documents into a structured form intended to be consumed by language models.

REVIN uses direct text extraction, because the reports it processes are text-based PDFs. What I saw matches what this literature reports: the choice of library visibly changed how well the tables of the report survived the conversion, and the third block ends up reading a Markdown view rather than a plain text one, as explained in Chapter 5.

2.4.1 Deciding first whether a document needs OCR

One recent tool addresses a problem REVIN currently leaves open. The pdf-inspector library (Firecrawl, 2026) does two things. First, it **classifies** a PDF by inspecting the internals of the

file, without rendering it, and reports which of four kinds it is: a document whose text can be read directly, a scan of paper, a document made mostly of images, or a mixture of the two, in which some pages carry text and others do not. Second, for documents of the first kind it converts the text to Markdown locally, in well under a second per document, without using optical character recognition at all.

The motivation its authors give is economic. A large share of the PDFs that systems process in practice contain real text and do not need OCR, so deciding first and routing accordingly avoids paying an OCR service for work it does not need to do.

The classification is the more immediately useful of the two. REVIN currently discovers that a report cannot be read only after trying to extract its text, and it then reports the outcome and stops, as described in Section 3.3. A classifier of this kind would let the system know in advance which reports require OCR, and, more valuably, it distinguishes the mixed case: an appraisal report whose main body is text but whose annexes, the land registry note or the cadastral record, arrive as scans. Those are exactly the documents REVIN handles worst today, because the extraction succeeds and yet part of the information is silently absent.

The local conversion is attractive for a different reason, one consistent with the cost argument that runs through this thesis: it does the ordinary case quickly and on the company's own machine, leaving any paid service for the documents that genuinely require it.

One thing this tool would not solve is the photographs of the interior and exterior of the property, which the appraiser includes as evidence of having visited it. They are images and not text, and neither this library nor OCR can tell whether such a photograph shows the property it claims to show. That would require a model that interprets images, which is a different problem and one this thesis does not address. Evaluating pdf-inspector on the project's own corpus is proposed as future work in Chapter 9.

2.5 RegTech and automated regulatory compliance

The use of technology to support compliance with regulation is known as RegTech. Arner et al. (Arner, Barberis, & Buckley, 2017) describe how, after the 2008 financial crisis, the growth in regulatory requirements pushed both regulators and firms towards technology for monitoring, reporting and compliance.

Micheler and Whaley (Micheler & Whaley, 2020) examine the more ambitious version of the idea, which is to replace rules written in legal language with computer code, and they set out its difficulties. Legal rules often contain deliberate ambiguity and require judgement, so translating them into code means making interpretative choices, and those choices then

disappear from view inside a system.

The review of appraisal reports is a compliance task of this kind, and the difficulty these authors describe appeared in this project in a concrete form. Turning a requirement such as "the surface areas must agree" into code forces a decision about what *agree* means, and Chapter 7 reports that the threshold was fixed by an explicit decision rather than assumed.

REVIN's answer to the general problem is structural. The rules live in code, separate from the language model, so every outcome is deterministic and can be traced back both to the rule that produced it and to the evidence in the document that the rule used. The interpretative choices therefore remain visible and can be reviewed, instead of being absorbed into a model's judgement.

2.5.1 Regulation is now catching up with the tools themselves

Until now the technology has served compliance with regulation. Recently the direction has reversed, and the technology itself is now regulated.

Regulation (EU) 2024/1689, known as the Artificial Intelligence Act (European Parliament and Council of the European Union, 2024), classifies artificial intelligence systems by the level of risk they carry. Systems placed in the high-risk category must meet obligations of transparency, traceability and human oversight, and that category includes systems used to assess whether a person is creditworthy. The obligations for high-risk systems became applicable in August 2026.

Whether a tool that reviews appraisal reports falls inside that category is a legal question this thesis does not try to settle, and the answer would depend on how the tool is used within the lending process. The requirements the regulation emphasises are the ones REVIN's design was already built around, for reasons that had nothing to do with the law and everything to do with being able to defend a result to a client. No decision is taken by the model, every outcome can be reproduced, every value carries the literal text and the page it came from, and the final judgement stays with a human reviewer. A design that keeps the model out of the decision is easier to examine under rules of this kind than one that does not.

2.6 Positioning of this work

Table 2.1 summarises what REVIN takes from each of these lines, and where it departs from them.

Table 2.1: What REVIN takes from each line of work.

Line	What it provides	What REVIN does with it
Property valuation	Automated estimation of value (Stang et al., 2022)	Not used: REVIN reviews an existing appraisal, it does not value
Generative extraction	Extraction without training per field (Brown et al., 2020; D. Xu et al., 2024)	One prompt per appraisal company, JSON output, no examples
Structured output	Guarantees of well-formed answers (Geng et al., 2023)	The provider's JSON mode, schema kept only in the prompt
Attribution	Verifiable grounding of statements (Rashkin et al., 2023; Gao et al., 2023)	A stricter rule: a literal quotation, checkable by a program
Long contexts	Performance falls as the input grows (Liu et al., 2024)	The prompt states where each section lives in the report
Document parsing	Benchmarks and toolkits (Li et al., 2024; Ouyang et al., 2025; Livathinos et al., 2025)	Direct extraction of the text. Sorting scanned reports from readable ones is left as future work
RegTech	Rules as code, and its limits (Arner et al., 2017; Micheler & Whaley, 2020)	Rules in code, separate from the model, with traceable outcomes

The gap this work sits in is a narrow one. There is a large body of research on extracting information from documents with language models, and a separate body on automating regulatory compliance, but comparatively little that joins the two on a document governed by a specific national regulation, where the extraction must be auditable because a financial decision depends on it. REVIN is an engineering answer to that combination, developed under the constraints of a real company, and its contribution is best read as a design and its justification rather than as a measured result.

Chapter 3
Solution Overview and Methodology

Chapter 1 set out what the review of an appraisal report involves, and Chapter 2 placed that task with respect to existing work. This chapter describes the system that was built to carry it out, and it comes before the detailed implementation of Chapter 5 because several of the decisions explained there only make sense once the overall design is clear.

3.1 A pipeline of four blocks

REVIN is organised as a pipeline of four independent parts, which I will call blocks. Each block does one job and hands its result to the next, so that a document moves through them in order. The first block receives the PDF and gives it an identifier, the second turns the document into text, the third uses a large language model to convert that text into structured data, and the fourth applies the checks and produces the result that a reviewer will read.

This is more than a division on paper, because the blocks never call each other. They communicate only through files kept in a shared working folder, with one folder for each document, which means that any block can be run, tested or deployed on its own without the others being present. Running the system from beginning to end is then the job of a separate piece, the orchestrator, which calls the blocks in order and passes the document identifier from one to the next (Section 5.5). Figure 3.1 shows the whole arrangement.

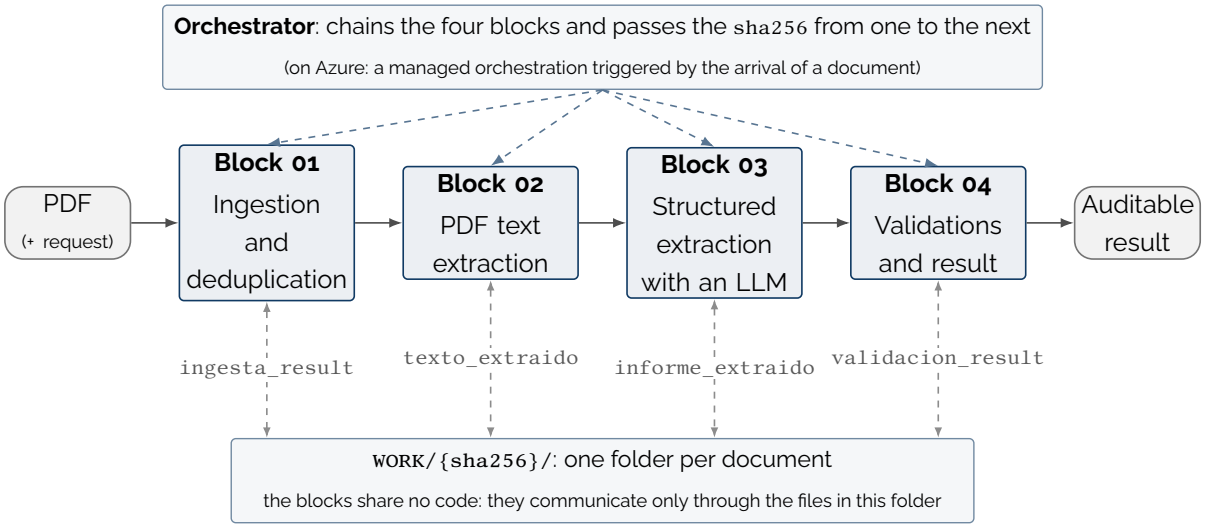

Figure 3.1: The REVIN pipeline. A PDF enters the first block and a reviewable result leaves the last one. All four blocks work on the same per-document folder, and an orchestrator chains them.

3.2 Design principles

Five principles were kept across the whole system, and they explain most of what follows.

- **Independent blocks.** Each block is a small, self-contained project that shares no code with the others and exchanges only files with them.
- **A content-based identifier.** Each document is identified by the SHA256 hash of its content, so the same PDF always receives the same identifier. Duplicates are then easy to detect, and any result can be traced back to the document that produced it.
- **Extraction and validation are separated.** The language model only extracts data and never decides whether the report is correct, because the rules are applied afterwards, in code, by the fourth block.
- **Every value carries its evidence.** Each extracted value keeps the literal text it came from and the page where it was found, so that any result can be checked without reading the original PDF.
- **Configuration by environment.** Paths and parameters are read from a single configuration file, so moving to the cloud becomes a matter of changing settings rather than code.

3.3 The contract of states between blocks

The blocks are held together by one agreement: **every block ends by returning a state**, and never by breaking. That agreement is what makes the chain possible, and I chose it with the move to the cloud already in mind.

A program normally reports how it went in one of two ways. Either it succeeded, or it failed with an error. The first version of REVIN worked like that, and the distinction turned out to be too coarse for the documents this system receives, because a report can end up in three situations rather than two and only one of them is a malfunction.

The first is the ordinary case, in which the document is ready for the next step. The third is a genuine failure, where something in the system has gone wrong. The one in between is the interesting one, and it happens more often than might be expected. The document is perfectly well formed and the system has understood it correctly, and yet it cannot be processed, as happens with a report that arrives as a scan of paper, or with one produced by an appraisal company whose template has not been studied yet.

A design that knows only success and failure has to push that middle case into one of the two, and neither choice works. Calling it a success hides a document that was never reviewed, while calling it an error raises an alarm about a situation that was foreseen and in which nothing has actually broken. REVIN therefore groups the states of every block into three families, which are listed in full in Appendix E.

- **States that continue.** The block did its part and the next block has what it needs, so the document goes on.
- **States that stop for an expected reason.** The document cannot go further and the system knows why, so the reason is recorded and reported.
- **States that stop because of an error.** Something failed, the state names what it was, and the document can be run again once the cause is fixed.

Not every block needs all three. The middle family appears only where a document can legitimately be impossible to process, which in practice means the second block, for a scanned report, and the third, for an unsupported appraisal company. The other two blocks have only states that continue and states that fail.

When a document stops for either of the last two reasons, the path of that document ends, but the run itself does not. Continuing would be pointless, since the following blocks would have no input to work on and would only produce a second, less informative state. What the

design really avoids is a block that breaks, because a block that breaks leaves the pipeline with no answer at all about what happened to that document.

This contract is also what allows the same code to move to the cloud without being rewritten. In Microsoft Azure the managed orchestration reads the state each step returns and decides whether the chain goes on, exactly as the local orchestrator does. An earlier version of the system reported its outcome through process exit codes, which carried far less information, because a number cannot say *why* a document stopped and leaves the caller to translate it back into a business meaning. Those exit codes were removed, and the state is now the only signal.

3.3.1 Idempotency and safe re-execution

A block can also be run again on a document it has already processed without repeating the work. When that happens it reports an "already done" state and returns the previous result. This matters in the cloud, where an orchestration may retry a step after a temporary failure, and it matters for cost, because it prevents the language model from being called twice for the same document.

3.4 The extraction contract: one prompt per company, and verbatim evidence

The third block is guided by a system prompt, which is simply a set of written instructions telling the model what to extract and how. Since each appraisal company uses its own report template, and the same value, the cadastral reference for instance, sits in a different place in each of them, the system keeps one prompt per company, written after studying that company's reports, rather than a single prompt meant to cover them all.

The central rule of that prompt is "verbatim or nothing". For every value, the model has to copy the exact text from the document and report the page where that text appears, and if it cannot copy the text literally then the value is left empty. This is the main defence against hallucination, and it is checkable: because the quotation has to be an exact substring of the document, a program can confirm that the value really was in the report.

3.5 Data flow and artefacts

A single document leaves a trail through the system. When the document arrives, the first block creates a folder named after its identifier and leaves inside it the PDF together with a small metadata file recording the original name. The second block reads that PDF and writes the extracted text in Markdown format, which is what the third block will read. The third block calls the model and writes the structured data as a JSON file, and the fourth block reads that file, runs the catalogue of checks and writes the result of each one.

The folder therefore grows as the document advances, and every step leaves something behind on disk. That is deliberate, because it means the state of a document can be inspected at any point of the process without having to run anything again.

3.6 Development methodology

The project was developed with a specification-first method. Before writing the code of a block I wrote a short specification for it, a plan and a list of tasks, which kept the goal of each block clear and reduced the amount of work that had to be redone later.

The project was also managed with an agile method, using Jira, and source control was handled with Git. Chapter 4 describes how that board was organised, and what it cost to keep it true in a project whose design was redirected twice.

3.7 Use of artificial intelligence during this work

Artificial intelligence was used as a working tool in this project, and the guidelines of the University of Alicante ask for that use to be described in the methodology.

3.7.1 During the development of the system

The code of REVIN was written with the assistance of Microsoft Copilot (Microsoft, 2026), which was the assistant the company made available to its data team. I used it as a writing aid, describing what a component had to do and then reading, correcting and adapting the code it produced.

What it did not do is decide anything. The design decisions of this project came from three places.

- **Given by the company.** That the validations had to be written in code and never handed to the model was a requirement from the start, for the two reasons stated in Section 1.4, and it is the principle the whole system rests on. The architecture of independent blocks was asked for by my tutor and designed together with him.
- **From my own research.** The choice of the content hash as the identifier of a document, and the choice of the libraries that turn the PDF into text, came from studying the available options and comparing them against what the pipeline needed.
- **Mine.** Converting the document page by page so that every page carries its number, the rule that every extracted value must carry a literal quotation, the three families of states described in Section 3.3, and the decision to keep the data schema in a single place instead of duplicating it in the code.

None of them came from the assistant, and neither did anything that was discarded, which in this project was considerable, since the whole of Chapter 7 is the record of designs that were built, used and then replaced. An assistant will write whatever it is asked for. It will not tell you that what you asked for was the wrong thing to build. Learning to tell the difference, mostly by getting it wrong first, is what the internship actually taught me.

This was my first project using object-oriented programming in earnest, so a good part of the internship went into understanding what the code I was directing actually did, tracing the errors that appeared and rebuilding parts of the system as my understanding of the problem improved. Working this way moved the effort from typing to reading, deciding and correcting.

I also ran into a limitation. An assistant produces code that works, which is not the same as code the project needs. The clearest example is the call to the language model described in Section 5.3.4, where what I had built was correct and far more elaborate than the problem required, and it took a person, my company tutor, to show me the short version that replaced it. That episode, discussed in Section 7.9, is the best argument I can give for why the judgement about what to build has to stay with the developer.

3.7.2 During the writing of this thesis

The drafting and revision of this document were assisted by Claude (Anthropic, 2026), used to structure chapters, to review the English and to check the text against the project's own documentation. The content, the decisions reported, the interpretation of what happened and the conclusions are mine, and every technical statement about the system was checked

against the project's specifications, its planning documents and its source code. No text was included without being read, judged and, where necessary, rewritten.

Chapter 4
Planning a Project Whose Design Kept Moving

The interesting question about the planning of this project is not which tool I used. It is what happens to a plan when the thing being planned is rebuilt twice while the plan is in force.

This chapter describes how the work was organised, and then examines what the planning actually cost and what it was worth. The detail of the board itself, the stories and the tasks, is in Appendix A.

4.1 How the work was organised

The project was managed in Jira as a single EPIC covering the whole system, divided into user stories and tasks, and it followed the specification-first method described in Section 3.6: for each block a short specification, a plan and a list of tasks, written before any code.

I made one decision about the board's structure that turned out to matter more than I expected: I mapped it onto the architecture. One story per block of the pipeline, plus a transversal story describing the whole, and one task per concrete piece of work inside each block.

- **H0, overview.** A transversal story describing the complete pipeline and the principles shared by the four blocks.
- **H1, ingestion and deduplication.** Receive the PDF, identify it by its hash, copy it to its working folder.
- **H2, text and table extraction.** Produce the text views of the document for the next block.

- **H3, structured extraction with a language model.** Turn the report into structured data, with evidence for every value.
- **H4, validation and result.** Apply the checks in code and write the result.

Mapping the board onto the architecture made the state of the work legible at a glance: anyone could see which part of the system was finished and which was not. It also had a consequence I had not anticipated. *If the board mirrors the architecture, then every change to the architecture is also a change to the board.*

4.2 What each kind of change cost

The plan was not fixed from the start. It changed in three ways, and they did not cost the same.

Work that appeared. Story H1 grew two tasks that had not been foreseen: copying an optional request file that may travel with the PDF, and an option to remove the input PDF once it has been processed. This is ordinary and cheap: the plan was incomplete, and completing it costs the price of two new cards.

Work that changed shape. Story H4 moved from a skeleton with no implementation to a first working version with a catalogue of checks taken from the control spreadsheet. The tasks did not disappear, they were filled in. This is also ordinary, and it is what a plan is supposed to accommodate.

Work that stopped being true. Story H3 is the different case. Its tasks described a design, the single general prompt with typed data models, that I later abandoned for the reasons given in Section 7.5. When the design went, the tasks did not need updating. They needed deleting. I rewrote that part of the board from scratch to match the new design.

Work that stopped being true is the expensive kind, and the kind a project like this produces repeatedly. Each of the reversals reported in Chapter 7 had a cost in code, which is visible, and a second cost in planning, which is not. Rewriting the board took time that did not advance the system by a single line.

4.3 What I would defend, and what I would not

I kept the board updated as the design evolved, rather than letting it drift into a description of a project that no longer existed. That was the right call, and it is why this thesis can reconstruct what happened: Appendix A and the timeline below come from the planning documents written while the work was happening, not from memory. A plan that is abandoned the moment it becomes inconvenient leaves nothing behind.

What I would not defend is how I used the board. I updated it after doing the work, as a log, more often than I planned with it beforehand. A record tells others what has happened, but a plan tells them what is about to happen, and only the second is useful to anyone but its author. In a company, where several people depend on knowing where a project stands, that is not a stylistic preference. It is the point of the tool, and it is the professional habit I identify in Section 7.14 as the one I most need to build.

The specification-first approach and a board mirroring the architecture pull in opposite directions when a design is unstable: the first assumes you know what a block should do before you build it, and the second makes every change to that assumption expensive to record. They worked together here because the specifications were deliberately short, a page or two per block rather than a full design document. Had they been longer, the cost of each reversal would have been high enough to tempt me into not reversing anything, which is the worst outcome of all.

4.4 Project timeline

Table 4.1 summarises the milestones. Read alongside Chapter 7, the table shows how close together the two big changes of direction fell: the split into blocks and the new approach in the third block were two weeks apart, in an internship of five.

Table 4.1: Main milestones of the project (2026).

Date	Milestone
18 to 21 May	Project start. The flow is designed as a diagram before any code: six phases, twenty-six nodes, inputs and outputs per node. First implementation: a monolithic, step-based pipeline with random identifiers.
21 May	Project meeting. The control spreadsheet is defined as the source of the checks.
24 May	First working prototype of the monolithic pipeline, with an interactive flow diagram.
28 May	**Change of direction.** At my company tutor's request, the system is rebuilt as four independent blocks that communicate through the content hash, following a specification-first method.
Early June	Blocks 01 and 02 completed. The dual text/Markdown view and its problems are settled.
2 June	Block 03 in its first "by source" design: a single general prompt with typed data models.
11 June	**Change of direction.** Block 03 rebuilt around one prompt per appraisal company, and the typed models are removed.
12 June	Block 04 reaches a first working version, with a catalogue of 43 checks taken from the control spreadsheet.
Mid-June	The third block is connected to the real cloud API and produces a correct extraction on the first run. An orchestrator is added that chains the four blocks.
Third week of June	The call to the model is replaced by the much shorter version supplied by my company tutor. The remaining days go into adding as many checks as possible to Block 04.
22 June	Last day of the internship. The system, and the results it produces on real reports, are presented to the team that will carry the work forward.

Chapter 5
System Implementation

This chapter describes the four blocks of the pipeline and the orchestrator that runs them as a chain. The pseudocode of each block is in Appendix F.

5.1 Block 01: ingestion and deduplication

5.1.1 What it does

The first block receives the name of a PDF placed in the input folder. Before anything else it checks three things: that the name is safe, that the file exists, and that it really is a PDF, which it confirms by the extension and by the first bytes of the file.

The first of those is a security check rather than a validation of content. The block is given a file name, not a full path, and it will join that name to the input folder to find the file. If the name contained a path of its own, something like a slash or two dots meaning the folder above, the resulting path could point outside the input folder altogether and the block could be made to read a file it should never touch. This is known as a path traversal, and the check rejects any name that is not a plain file name, returning `ERROR_NOMBRE_INVALIDO`. It costs nothing, and it stops the problem before it can arise.

Once the name is accepted, the block computes the SHA256 hash of the content, which becomes the identifier of the document. If a folder for that identifier already exists, the document was processed before, so the block stops without copying it again. If the document is new, the block creates its folder, copies the PDF keeping its original name, and writes a small metadata file. The block can also copy an optional request file that travels with the PDF, and it can delete the input PDF after a successful copy.

Figure 5.1 shows this flow, and Pseudocode F.1 sets out its logic step by step. The same colour code is used in the four diagrams of this chapter: green for a successful end, grey for a

controlled end that is not a failure, and red for an error state, following the three families of states of Section 3.3.

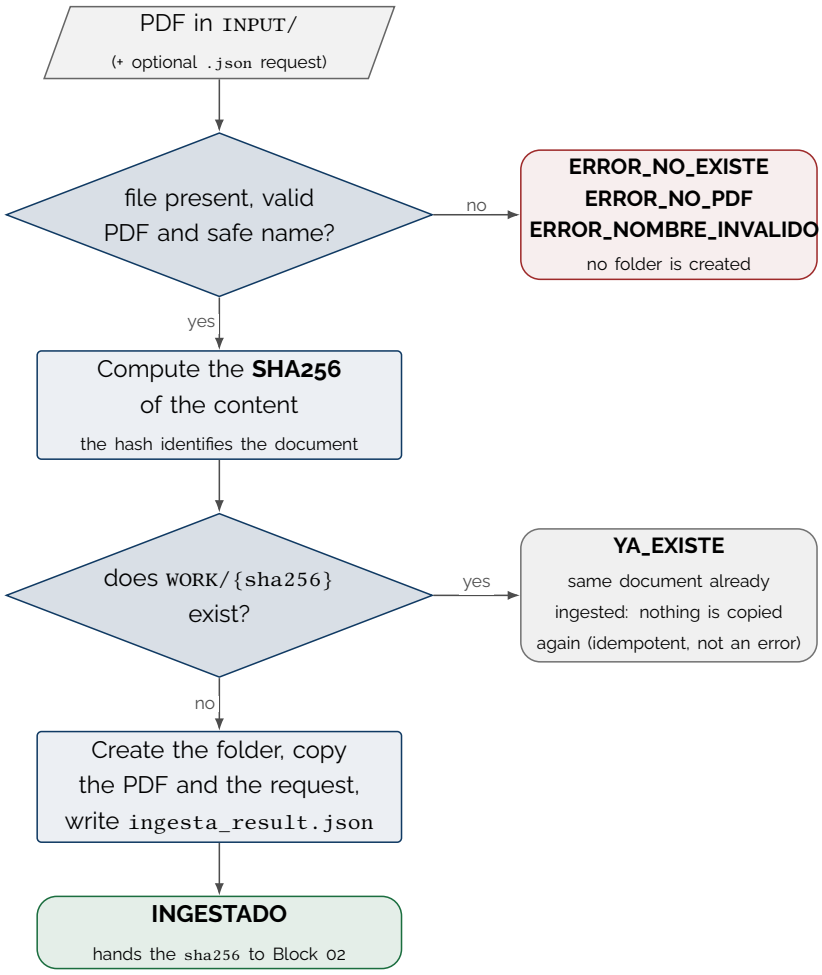

Figure 5.1: Block 01, ingestion and deduplication.

5.1.2 Problems I ran into

Two features were added that were not planned at the start: copying the optional request file that may travel with the PDF, and the option to delete the input PDF after a successful copy. The deletion was made safe: the PDF is removed only after the copy in the working folder is confirmed, and never when there is an error.

The identifier itself also changed during the project. An earlier version gave each document a random identifier, which was replaced by the content hash (SHA256), so that the same PDF always maps to the same folder. This change is discussed in Chapter 7.

5.2 Block 02: PDF text extraction

5.2.1 What it does

The second block reads the PDF and produces two views of its content: a plain text view and a Markdown view. The plain text view keeps the text page by page. The Markdown view keeps the structure of the tables better, and it is the view that the third block reads. The block also writes a metadata file with the number of pages and how much text each view obtained. If one view fails, the other is still written, so a single failure does not stop the block. If neither view obtains any text, the document is marked as unreadable, which is the case of fully scanned PDFs.

5.2.2 Why the block writes the text twice

At the start the plan was to keep only the plain text. That view was good for reading the document page by page, but it did not keep complex tables well. The Markdown view kept them in a clearer form, and for this reason the third block reads only the Markdown view, which gives a single source for the evidence and avoids mixing two versions of the same document.

The two views are produced by two different libraries, although that difference is smaller than it looks. The one that produces the Markdown relies underneath on the same PDF reader as the plain-text one, so the two see the document in much the same way. They differ in how much of that reader each one lets me control, and Section 5.2.3 shows why that mattered.

Two ways of failing that are easy to confuse. Having two branches also made it necessary to separate two outcomes that sound alike. ERROR_PDF means the file could not be opened and read at all, because it is corrupt, password-protected or not really the format it claims to be. Nothing was obtained because nothing could be attempted. PDF_ILEGIBLE is the opposite situation: the file opened perfectly, both branches ran to the end, and they came back with no text. That happens when the report is a photograph of paper rather than a document with text inside. The first is a failure of the system to read a file. The second is an accurate report about a file that has nothing to read. That is why one of them is an error and the other is not.

Figure 5.2 shows this flow, and Pseudocode F.2 sets out its logic step by step.

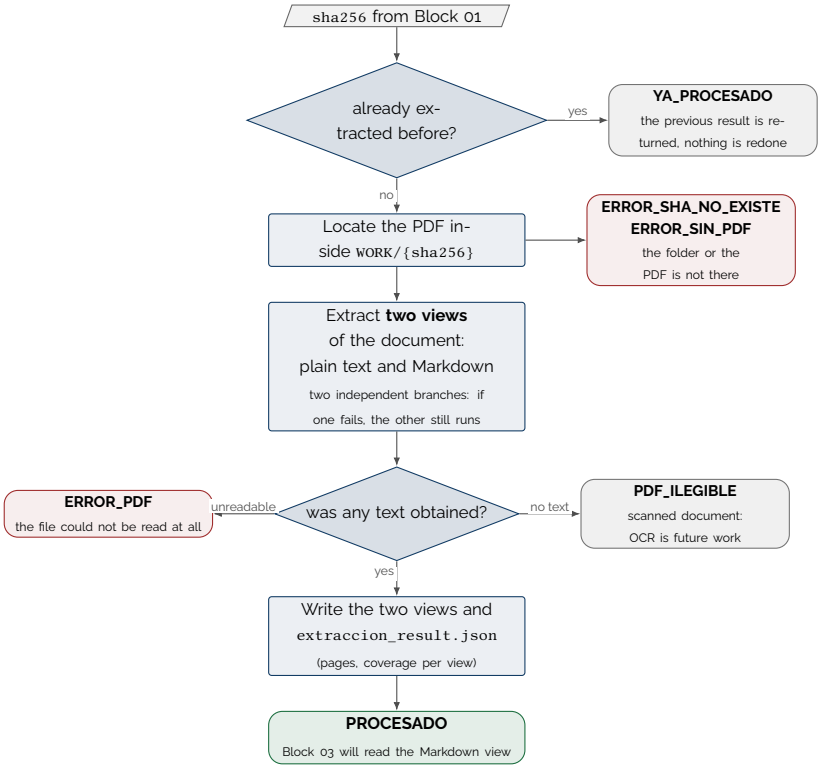

Figure 5.2: Block 02, PDF text extraction.

5.2.3 Problems I ran into

Two problems appeared in this block. First, the extraction of tables in the plain text view never worked perfectly: to avoid duplicating them, the tables are moved to the end of the page, which splits the information. This was one of the reasons to read only the Markdown view in the next block.

Second, both views came out dirty, and for the same reason. The reports carry a vertical digital signature stamp along the side of the page, and because the two libraries read the PDF in much the same way, that stamp appeared in both outputs as columns of doubled, reversed letters. It had to be removed twice, and the two solutions are not alike.

For the plain text the reader itself provides the answer. Every character it returns carries a flag saying whether it is upright, so asking it to keep only upright characters removes the stamp cleanly, at the level of the character and before any text is assembled.

For the Markdown that option does not exist. The library that produces it does not expose it, and I did not want to modify the library, since doing so would mean giving up the updates that come with it. So I wrote the cleaning step myself, outside it, working on the finished text.

Spotting the stamp was easy. The hard part was avoiding damage to the real text, because a line of one or two characters looks like part of a stamp and sometimes it is simply a short line of the report. My cleaning step removes a run of suspicious lines only when there are at least five together, and it keeps a normal line that happens to be surrounded by them. Converting the document page by page, which was needed anyway for the page markers, helped here too, because the stamp is then confined to the page it belongs to.

The same problem cost a single instruction in one branch and a small piece of reasoning of my own in the other, because each library allowed me to reach a different amount of the underlying reader.

5.3 Block 03: structured extraction with a language model

5.3.1 What it does

The third block is the centre of the system. It reads the Markdown view, detects which appraisal company produced the report, loads the prompt written for that company, and calls the language model to obtain the data as a JSON object. It writes two files: the JSON data, which the fourth block will read, and a small work record with the state, the company, the model used and any warning.

One of those warnings is the only check the block makes on the answer. After receiving the JSON, the block compares the sections it contains against the ones the prompt asked for, and writes down any that are missing. It does not reject the answer and it does not correct it. An incomplete extraction simply should not pass unnoticed, and noticing costs nothing.

5.3.2 Working out which company wrote the report

Before anything else the block has to know which appraisal company produced the report, because the prompt it will use depends on that. This is done in ordinary code, with no help from the model, by looking for three clues one after another and stopping at the first that works: the original name of the file, the request file if one came with it, and finally some marks in the text of the document itself. If none of them matches, the block stops there and says so.

I put this check first on purpose. The model is the only part of the system that costs money every time it runs, so anything that can be decided without it should be decided before it. A report from a company we do not support then costs nothing at all, because the expensive call simply never happens.

There was time to study only one appraisal company during the internship, so in practice this was the outcome for every other one. I think that is the right behaviour rather than a gap. A client can always send a report from a company nobody has looked at yet, and saying so plainly is more useful, and a good deal cheaper, than returning an extraction I know would be poor.

5.3.3 What the model gives back

The JSON produced by the model is organised into eighteen blocks, one for each main section of the report, such as the cover, the surfaces, the registry note and the cadastral data. Each value is stored together with its literal quotation and its page. The schema is described here at a conceptual level, because the exact instructions given to the model are part of the prompt, which is not reproduced.

5.3.4 The three ways of getting an answer

The block obtains the JSON through one of three interchangeable options, chosen in the configuration file: the commercial cloud service, a model running on a local machine, and a stub that answers without a network. They all offer the same operation, so the rest of the block does not know which one is answering. Only one of them is in real use, and cost is what put the other two there.

The cloud provider, which is the one in use. The call itself is short: it sends the system prompt of the detected appraisal company and the Markdown of the report, asks the API to answer in JSON, and sets the randomness of the model to zero so that the same document gives the same answer. This is the shape the call reached at the end of the internship. An earlier version of mine was considerably more elaborate. My company tutor showed me a much shorter way of doing the same thing, and replacing it simplified a large part of the block. It was a useful lesson, and it is discussed in Chapter 7.

The stub provider, which I wrote and never really used. A stub is a fake implementation that returns a fixed, canned answer and needs no network. I wrote one because the company's API key was not available to me for most of the development, and because every real call has a price, so the block needed some way of starting up without spending anything. In practice it never became part of how I worked. The prompt was refined by hand against the assistant the company provided, as explained in Section 7.7, and the block itself was only

really exercised once it was connected to the real service. The stub stayed in the code as a scaffold rather than as something I developed against.

The local provider, which is unfinished. Running the model on the company's own hardware was explored because of a concern raised in meetings: the cost of commercial language models is a recurring external expense, and the company was considering whether, in the coming years, moving to its own servers would be cheaper than paying per call. I prepared the block for that possibility, so that switching would be a change in the configuration rather than a rewrite. The attempt did not get far on the hardware available to me, for the reasons described in Chapter 7.

Of the three paths, only the cloud provider was ever really used. The local one was left unfinished when the cloud call was rewritten in its short form near the end of the internship. All three remain selectable in the configuration, but the default, and the only path actually in use, is the cloud provider. Section 7.14 discusses why this part of the block is the one I would now design differently.

An earlier version of the block recorded the number of tokens of every call, which is the figure a cost analysis would be built on. That accounting went out with the simplification described above, and the block as it stands does not measure what a document costs to process. Putting it back is a small change, and Chapter 6 explains why it is the first thing I would restore.

5.3.5 What I learned from reading the reports

The prompt was written after studying a set of real reports from one appraisal company, and that reading is where several of the decisions in the schema came from. Some annexes arrive scanned and cannot be read as text at all. The page number printed at the foot of the report does not match the real order of the pages, which is why the block adds its own page markers and the prompt is told to use those and never the printed ones. And wherever the report states something that could also be worked out by counting or adding up, the prompt asks the model to copy what is written rather than produce the figure itself.

Figure 5.3 shows this flow, and Pseudocode F.3 sets out its logic step by step.

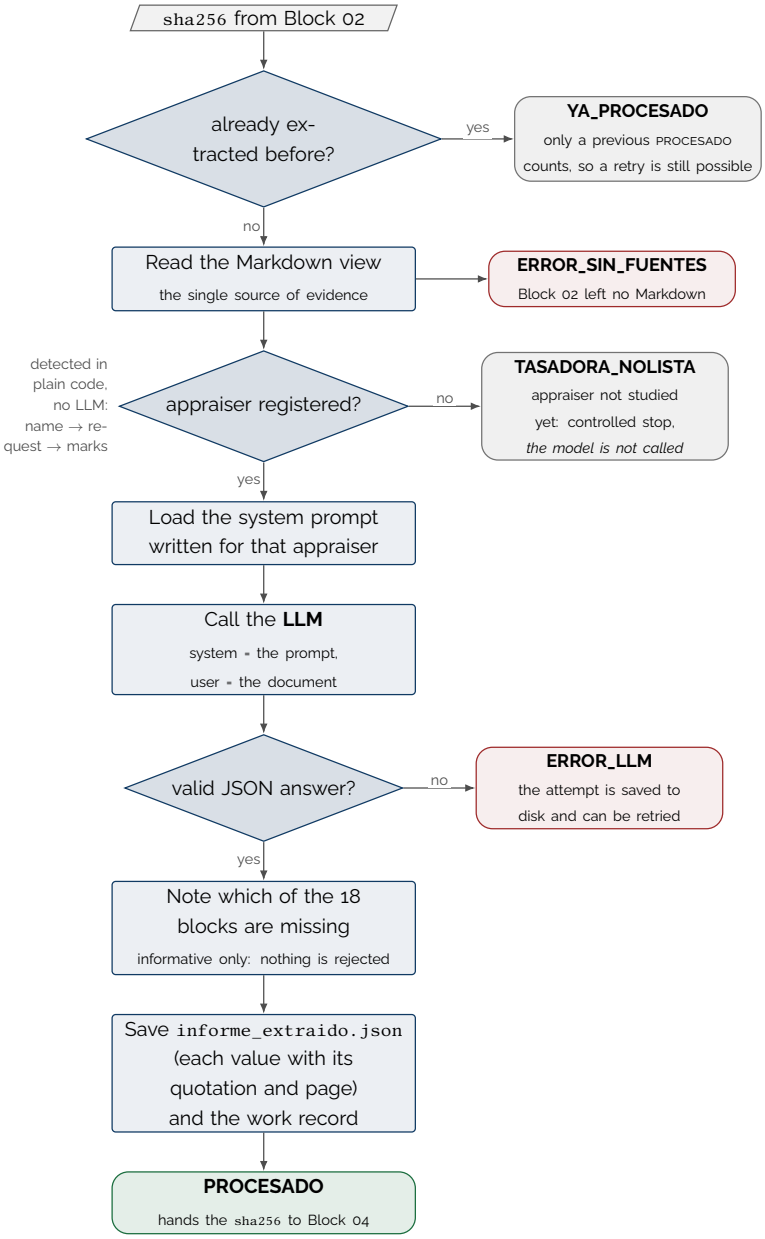

Figure 5.3: Block 03, structured extraction with a language model. The appraiser is detected in plain code, so an unsupported document never reaches the model.

5.3.6 Problems I ran into

This block changed the most during the project. The first design used a single general prompt that read both views of the document and returned the data grouped “by source”, with a large set of typed data models in the code to hold the result. This approach was abandoned: a

general prompt did not know where each value lives in a particular company's template, the schema had to be kept in two places (the prompt and the models), and data from different sections mixed together. It was replaced by one prompt per appraisal company, and the typed models were removed so that the schema lives only in the prompt.

Running the model on a local machine was explored and set aside, for reasons of hardware that are examined in Section 7.11.

5.4 Block 04: validations and result

5.4.1 What it does

The fourth block reads the JSON data produced by the third block and, if it is present, the optional request file. It then runs a catalogue of checks written in plain Python, and writes a result file with the outcome of each check and a count by group. It uses no language model: all the rules are deterministic code. The block is generic, that is, it does not depend on the appraisal company, because it works on the common data format produced for every company.

5.4.2 The catalogue of checks

The checks are grouped by category, following the control spreadsheet provided by the company. They cover the purpose of the appraisal, the documentation, the registry note, the surfaces and the values, among others. Each check compares values, or confirms that a required element is present, and returns one of four outcomes: pass, fail, no data, or not applicable. Each check also keeps the values it compared, the evidence behind them, and a reference to the row of the spreadsheet, so the result can be traced. The catalogue described here has 43 checks in 18 groups, and it kept growing during the remaining weeks of the internship as the meetings with the team clarified further controls from the spreadsheet. A single global verdict is not produced, because the criterion for it was still being agreed when the internship ended.

Figure 5.4 shows this flow, and Pseudocode F.4 sets out its logic step by step.

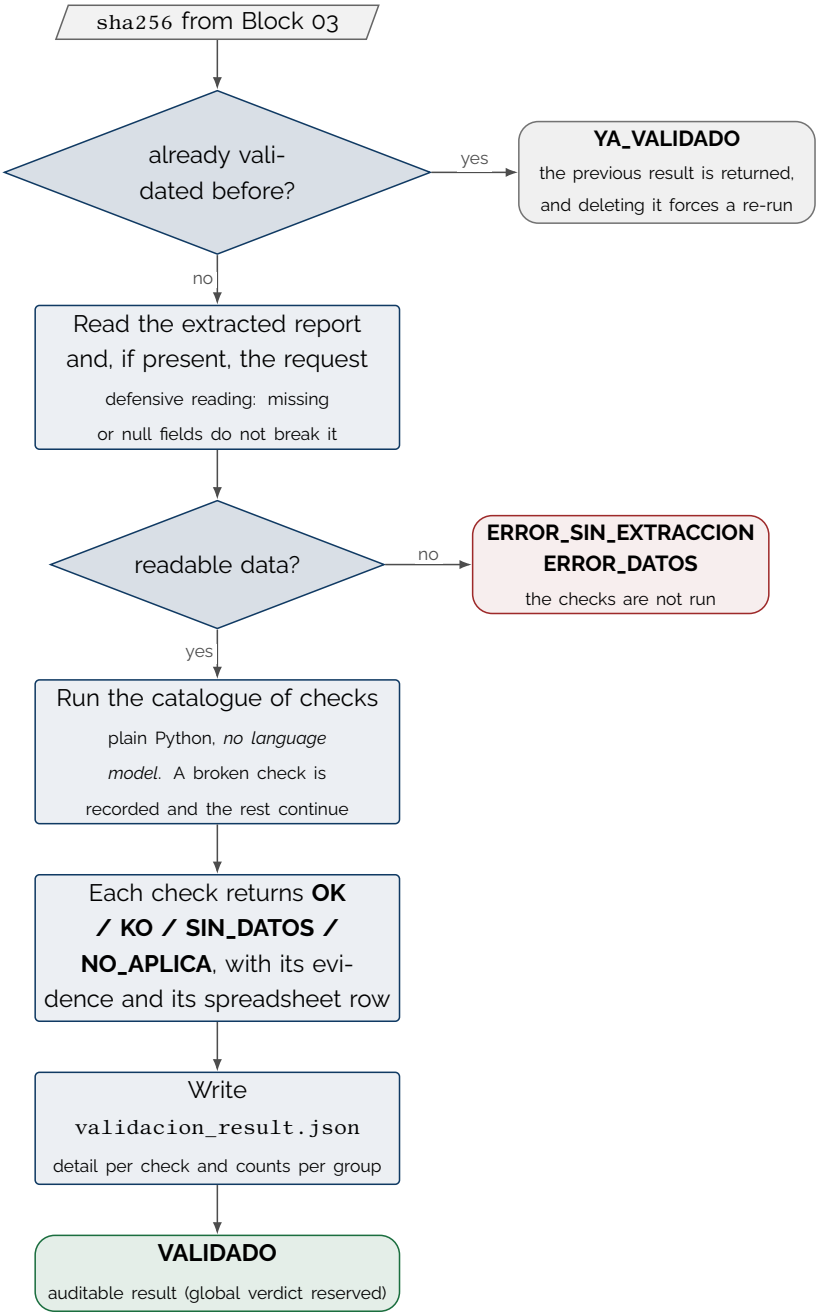

Figure 5.4: Block 04, validations and result.

5.4.3 Problems I ran into

This block went from an empty skeleton to a first working version with a catalogue of 43 checks. The control spreadsheet provided by the company was the source of truth: it decided

which checks to write and which values the previous block had to extract.

The main work was the careful, defensive reading of the JSON data and the normalisation of values, so that dates, numbers and references written in different forms can be compared. Some thresholds were fixed by an explicit decision rather than guessed, such as requiring the surfaces to match exactly. The result file keeps a field for a global verdict, but the verdict is not computed yet, because the final set of rules was not available during this work.

5.5 Running the blocks as a chain

At first, the four blocks were run one by one, by hand: I launched a block, read the identifier it printed, and passed that identifier to the next one. Later, an orchestrator was added to do exactly that automatically. It calls the blocks in order and takes the output of each one as the input of the next, so a document travels the whole pipeline in a single run: the PDF enters Block 01, and each block starts when the previous one has finished, until Block 04 leaves the result. The blocks stay independent and keep communicating through files, and the orchestrator only coordinates them.

The orchestrator works on the state contract described in Section 3.3. After each block it reads the state that the block returned and decides from it whether the chain goes on. A state of the continuing family means the next block has its input ready. A state that stops the document, whether expected or an error, ends that document's path there, because the following blocks would have nothing to work on. The state is recorded so that the outcome can be read afterwards, and the document can be run again later without repeating the work that was already done.

This piece is deliberately thin. It holds no business logic: it does not know what an appraisal report is, and it does not decide anything about the document. It only knows the order of the blocks and how to read a state, and that is what makes it replaceable, which matters for the next step of the project, described in Section 5.6.

Pseudocode F.5 sets it out in full. It is short next to the four blocks, because almost everything the system knows lives inside them.

This was also the point at which the third block was connected to the real cloud API, and at which more checks were being added to the fourth block.

5.6 Ready for the cloud

The whole system is prepared to run on the cloud, on Microsoft Azure. The orchestrator just described is, in fact, the shape the system will have there: on Azure, each block becomes a function, and a Durable Functions orchestration (or Logic Apps) plays the role of the runner that chains them, triggered by the arrival of a document. Because the orchestrator holds no business logic, this substitution does not touch the blocks. Only the caller changes, and the blocks do the same work as before.

Three properties of the design make that move possible. The blocks are independent and communicate through files, so each one can be deployed on its own. They can be re-run safely, so an orchestrator may retry a step after a temporary failure without duplicating work or paying twice for a call to the language model. And they answer with a state rather than with an exit code, so the managed orchestration can read the outcome of a step and branch on it directly, which is precisely what the exit codes of the first version could not express.

The actual deployment was not part of this work and is listed as future work in Chapter 9.

Chapter 6
Evaluation Design

Section 1.5 explained why this thesis reports no measured results. This chapter carries out the part of that work which remained possible: specifying, in enough detail to be executed, how REVIN should be evaluated.

A protocol is not a measurement, and I do not present it as one. Defining it still forces the same questions a measurement would have forced. What counts as a correct extraction? Which errors are expensive and which are merely annoying? How many reports would be needed, and what could make the resulting numbers misleading?

6.1 Two considerations before the protocol

The first consideration is why I give no approximate number instead of a real one.

I could have written something like “the system extracted around ninety per cent of the values correctly”. I did not, because I could not show where that number came from, and nobody reading this could check it. A number that cannot be checked still looks precise, and that is the problem with it: readers trust numbers more than they trust sentences, so an unchecked number would carry more weight here than it deserves.

There is also a reason particular to this project. The whole point of REVIN is that its conclusions can be verified, one by one, against the document. It would be odd for a thesis that defends that idea to end by asking the reader to take a number on trust. Giving no number is the honest option, and it is the one that matches what the system itself is for.

The second consideration is that, even with full access, part of this protocol could not have been executed during the internship. The priority set by the company was to reach a working pipeline connected to the real service, and the definitive set of validation rules was still being prepared when the internship ended. Evaluating the fourth block against rules that were about to change would have measured a moving target, which is why the third level

described below remains conditional on those rules being fixed.

6.2 What would have to be measured

REVIN is not one model but a chain, and a single accuracy figure would hide where its errors come from. The evaluation therefore has to be split into three levels, which differ enormously in what they cost to run.

6.2.1 Level 1: verifiability of the evidence

The cheapest measurement is also the one that tests the system's central design claim, and it needs no manual annotation at all. Every value the model extracts must carry a literal quotation from the document. Whether that quotation actually appears in the document is decidable by a program: it is a substring test against the extracted text.

Running that test over a set of reports gives two numbers directly. One is the proportion of extracted values whose quotation can be found in the document, which measures the "verbatim or nothing" rule of Section 3.4 head on. The other is the proportion of fields left empty, which says how often the model preferred silence to invention. Neither figure tells us whether the extracted value is the *right* one: a model can quote faithfully from the wrong part of the document. What this level establishes is the weaker but necessary property that the system is not fabricating text.

I would run this one first. It costs almost nothing, and a poor result here would sink the design rather than merely lower a score.

6.2.2 Level 2: accuracy of the extraction

The second level asks whether the extracted values are correct, and it is the one that requires human work. It needs a reference set: a sample of reports in which a person has written down, field by field, the value that should have been extracted. The system's output is then compared against that reference.

Comparing the two is less obvious than it sounds, because the same fact is written in different ways across a report. A date can appear as `12-05-2026` or as *12 de mayo de 2026*, and an amount as `150.000,00` or as *150000.0*. Comparing the raw text would mark those as wrong when they are not. The values therefore have to be tidied up first, and the fourth block already knows how to do exactly that, so the evaluation should use its rules rather than invent new ones. That way what is measured is the system as it really behaves.

For each field, three numbers should be reported separately rather than merged into one:

- **Correct**: the value was extracted and matches the reference.
- **Missing**: the reference has a value and the system left the field empty. The cost of this is reviewer time: a check downstream ends as `SIN_DATOS` and a person has to look the value up.
- **Wrong**: the system extracted a value that differs from the reference. That is the serious one, because a wrong value can make a check pass or fail for the wrong reason without anyone noticing.

Reported this way, the result is actionable: it identifies which fields of the prompt need work, which is exactly how the prompt was refined during development, only with numbers instead of impressions.

6.2.3 Level 3: agreement with a human reviewer

The third level measures the system as its users would experience it. A reviewer examines a set of reports by hand and records, for each of the checks in the catalogue, whether the report passes or fails. The system's verdicts are compared against those.

Here the two kinds of error are not equally bad. A **false pass**, in which the system reports a check as correct when the reviewer finds a defect, means a non-compliant appraisal can go through unnoticed, and the whole purpose of the system fails silently. A **false failure**, in which the system flags something the reviewer considers fine, costs the reviewer a few minutes of checking. In a review process a false pass is far more damaging, and this asymmetry should shape both the reporting and any threshold the system uses: recall over the defects matters more than precision, and a system biased towards flagging is preferable to one biased towards passing.

This asymmetry also explains a decision already taken in the design, that the global verdict was left unset. Turning several dozen individual outcomes into a single accept-or-reject requires knowing how expensive each kind of error is to the business. The evaluation would give how often each kind occurs, but what each one costs is a business judgement that stays with the company.

6.3 Corpus, sampling and annotation

The reference set has to be built with care.

Held-out reports. The prompt was written and refined against a set of real reports, which means it is fitted to them. Any evaluation on those same documents would report the quality of the fit rather than the quality of the system. The reference set must consist of reports that were not used while writing the prompt, and this is the single most important requirement of the protocol.

Sampling. The sample should be stratified rather than random, because the difficulty of a report is not uniform. It should deliberately include reports with scanned annexes, reports of property types that are less common, and reports where the registry note or the cadastral record is missing, since those are the cases that produce the empty fields the system has to report rather than conceal. About thirty reports would be enough to expose problems that repeat field by field, and that is all a first iteration needs. Telling apart small differences between models or prompt versions would take more.

Annotation. The annotation should be done by someone who reviews these reports professionally, not by the author of the system, for the obvious reason that the person who wrote the prompt knows where it looks for each value. A subset should be annotated twice, by two reviewers independently, and their level of agreement reported. That figure is informative in itself: fields where two experienced reviewers disagree are fields where the system cannot be expected to be unambiguously right either, and where the requirement, rather than the model, may be what needs clarifying.

6.4 Cost per document

The system cannot report the cost of a document at the moment. The third block used to record the number of tokens of every call, and that accounting went out with the simplification of the call described in Section 7.9. Restoring it means reading two numbers the provider already returns with every answer, so it is the cheapest item in this whole chapter.

With those numbers and the provider's prices, the cost of reviewing one report follows directly, and that figure is the one the company needs for two decisions.

The first is whether the system pays for itself, which is a comparison against the cost of the reviewer time it replaces. The second is the question raised in Section 7.11: at what volume of reports does buying hardware become cheaper than paying per call. Answering it requires the same reports to be run on a local model, which would give the throughput to set against the investment, and would at the same time provide a comparison of extraction

quality between a commercial model and a local one, using the metrics of Section 6.2. That comparison is the natural continuation of this work and is taken up in Chapter 9.

6.5 What the system would be worth

The previous section leaves that first comparison open, because the cost of processing one report cannot be produced today. The other side of the comparison, the reviewer time the system replaces, can be approached, and this section does so. What follows is a calculation over stated assumptions rather than a measurement, and it is written that way on purpose. It gives the company an order of magnitude and a rule for deciding, not a figure to quote.

Three assumptions carry the whole calculation, so each one is worth stating with where it comes from.

- **Reviewing a report by hand takes about fifteen minutes.** This is the estimate of my company tutor, who runs the data area and has watched the work being done for years. Nobody timed it, so it is an informed estimate and not a measurement.
- **Reviewing a report with the system takes about five minutes.** That is the time to read what the system has flagged and to look up the values it could not extract. This assumption is mine, and it depends on the extraction working well.
- **A reviewer costs 25,000 euros a year and works 1,720 hours.** The real cost to the company is higher, because a gross salary leaves out the employer's social security contributions, so taking the gross figure keeps the estimate on the conservative side.

At 25,000 euros over 1,720 hours a reviewer costs €14.53 an hour, which is €0.24 a minute. Fifteen minutes of review therefore cost €3.63 and five minutes cost €1.21, so each report reviewed with the system saves €2.42 of reviewer time, or €2,420 over a thousand reports. Read as throughput instead of as money, the same change takes one reviewer from four reports an hour to twelve, which is ninety-six in an eight-hour day instead of thirty-two. Adding the employer's contributions, at roughly 33,000 euros a year, raises the saving to €3.20 a report.

That saving is gross, because processing a report is not free either. Every document sent to the model is paid for, and Section 6.4 explains why that figure cannot be given today. The saving can still be turned into something the company can check, because it sets a ceiling. The system pays for itself as long as processing one report costs less than €2.42. That is a condition rather than a claim, and restoring the token accounting settles it in an afternoon.

Table 6.1: The saving per report under three sets of assumptions. The saving and the break-even cost are the same number, because the saving is exactly what the company can afford to pay for processing one report.

Scenario	By hand	With REVIN	Saving	Break-even
Conservative	10 min	8 min	€0.48	€0.48
Central	15 min	5 min	€2.42	€2.42
Optimistic	20 min	3 min	€4.12	€4.12

The times move the result far more than the salary does. Holding the central times and changing the salary to 22,000 euros gives €2.13 a report, and 30,000 euros gives €2.91, whereas the range across the three scenarios above runs from €0.48 to €4.12. Anyone who wants a firmer number should therefore time the reviewers before arguing about salaries.

Two limits qualify all of this. The five-minute figure assumes that the extraction works well, and that is precisely what this chapter says has not been measured, so the saving is conditional on the level-2 result of Section 6.2. A system that extracted poorly would send the reviewer back to reading the whole report, and the saving would disappear.

The calculation also counts only time. It says nothing about the cost of a false pass, which Section 6.2 identifies as the expensive error, because a defective appraisal that reaches the bank costs the company more than the minutes it saved. A system that was faster and let more defects through would be a worse deal, and nothing in this section would show it.

6.6 Threats to validity

Five limitations would qualify any result obtained with this protocol, and they should be reported alongside it.

- **One appraisal company.** Only one company was studied, so the figures would describe the system's behaviour on one report template. They would say little about how well the approach transfers, which is precisely the claim that adding a second company would test.
- **Prompt fitted to the corpus.** Even with held-out reports, the prompt was shaped by the patterns of one company's documents. Performance should be expected to drop on a new template, and by how much is unknown.

- **The reference is not ground truth.** It is one or two reviewers' reading of the documents. Where they disagree, there is no correct answer to measure against, only a range.
- **A moving target.** The catalogue of checks was a first version awaiting the company's definitive rules. A level-3 measurement is only valid for the version of the rules it was run against.
- **The cost model rests on assumptions.** The times and the salary of Section 6.5 are one informed estimate and a set of stated assumptions, not measurements. They give an order of magnitude and a decision rule, and they should be recomputed with the company's own figures before anyone acts on them.

6.7 What it would take to run this

The three levels are ordered by cost, and they should be run in that order. Level 1 requires no annotation and could be run in a day against the reports the company already holds. Level 2 requires perhaps thirty reports annotated by hand, which is a matter of days of a reviewer's time and is the step that would turn the prompt refinement from a qualitative process into a measured one. Level 3 requires the definitive validation rules, and it is the one that would tell the company whether the system is ready to be trusted.

None of the three requires anything that was unavailable for technical reasons. What they required was time inside the company and access to its data, and it is that access, rather than any property of the design, that this thesis lacked.

Chapter 7
Discussion: Design Decisions, Trade-offs and Lessons Learned

Several of the decisions described in this thesis were taken, used for some weeks and then undone. This chapter goes through them in the order in which they happened, because the reasons why my first answer was wrong are the part of the work that taught me the most.

7.1 The project began as a drawing

During the first days of the internship I did not write any code at all. I spent that time drawing the system instead.

On the first day I made a rough sketch of the flow in a diagramming tool, and on the second day I turned that sketch into a detailed version. It had six phases and twenty-six nodes, and for every node I wrote down what it received and what it produced, including the data types. The diagram soon became too large to read on a single sheet of paper, so I also built an interactive version of it as a web page, in which each node could be opened to show its inputs and its outputs. I only started sketching the classes that would implement the system after that map was finished.

Two of the decisions I took on that map survived every change that came afterwards.

The first decision was about where the language model would sit in the flow. I placed four checks written in ordinary code before it, so that the model appeared at a single node out of the twenty-six. That ordering is still present in the final system, in the way the appraisal company is detected before any call is made.

The second decision was to organise the pipeline as a sequence of independent steps, in which each step receives a piece of shared state and returns it. I chose that shape because I was already thinking about a possible move to a managed orchestration on Azure, where each

step would become an action. I later rewrote the implementation of that idea completely, but the shape itself did not change. The chain of steps that hand their state along is still how the final system works. The division into four independent blocks came later, and it came from my tutor, as Section 7.2 describes.

7.1.1 Where the working method came from

The specification-first method described in Section 3.6 also came out of those first two days. Writing down what a component receives and what it returns, before writing the component itself, had already saved me from several inconsistencies in the diagram, and it kept saving me from them later on in a project where I redesigned several parts.

7.2 From a monolith to independent blocks

My first working version was a single program that processed a document from beginning to end. The program worked, but the structure was wrong.

My tutor asked for the restructuring into four independent blocks, and we designed it together. There were two reasons for it, and neither of them had anything to do with elegance. The first reason is that independent blocks can be deployed one at a time on the cloud, which matters when the deployment is going to be gradual. The second is that each block can be changed without disturbing the others, which matters when parts of the system are still being redesigned.

I expected the restructuring to be painful, and it was not. The logic of each phase had already been written and was already working, so what changed was where each piece lived and how the pieces communicated with each other. The whole change took less time than I had feared. The fear of rewriting is often what keeps a bad structure in place, and in this case none of the work was thrown away. It was simply moved.

The principle underneath that split is that the model extracts and the code validates. It came from the company and was stated before I wrote anything, as explained in Section 1.4, and it turned out to be the decision that shaped everything else. It is the reason why the fourth block is the same for every appraisal company, why every result can be reproduced and defended in front of a client, and why the boundary of the language model's responsibility is fixed. Both Section 7.5 and Section 7.12 are consequences of that same principle.

7.3 Running the blocks together

Once the four blocks worked on their own, the next step was to run them as a chain, which is described in Section 5.5. The orchestrator is a short piece of code, so the interesting part of this change is not the code itself but what writing it forced me to make explicit.

While I was running the blocks by hand, the outcome of a block was something that I read and interpreted myself. As soon as a program had to make that decision instead, the possible outcomes had to be defined precisely, and it became clear that "it worked" and "it failed" were not enough. Two cases did not fit into either of those two labels: a report that arrives as a scan, and a report from an appraisal company that I had not studied yet. In both cases the system has understood the document perfectly and still cannot process it, and the pipeline has to be able to say so. That is where the three families of states of Section 3.3 come from, and it is also why the exit codes of the first version had to be removed.

The same change reopened an earlier decision about idempotency. I had designed the per-block "already processed" states when each block was still run separately, in order to avoid repeating work. Now that the model has a real cost, those states also avoid paying for the same call twice, and that is their main value today. When a document stops halfway through the chain, running it again retries only the step that stopped. Whether they will still be needed once the pipeline runs unattended on the cloud is an open question, and Chapter 9 discusses it.

7.4 Naming documents by their content

In the first version of the system I gave each document a random identifier, and the working folder was named after that identifier. I later replaced it with the SHA256 hash of the content of the file.

The decision came out of the restructuring described above. While I was deciding how the working folder would be organised, I realised that generating a random identifier was work the system was doing for nothing. I needed one unique name per document, and the document already contains something that identifies it uniquely. Using a hash of the content, which is a standard practice, meant one less thing to generate, to store and to keep consistent.

What I did not anticipate was how much would follow from that decision. Because the name comes from the content of the file and not from its name, the system recognises a document that it has already processed even when the file has been renamed, and files are renamed often when the same report circulates between departments. Duplicates are therefore detected

by the simple fact that the folder already exists, without any list of processed documents having to be maintained anywhere. Every duplicate detected in this way is also a document that is not sent to the language model a second time. A decision that I took in order to save myself effort turned out to be one of the clearest cost savings in the system.

The hash works in the other direction too, because any result can be traced back to the exact bytes of the document that produced it, and that matters in a system meant to be audited. The price of the decision is that a document edited in any way, however small the edit, becomes a different document for the system. For this particular use that behaviour is correct, since a modified appraisal report really is a new report and has to be reviewed again.

7.5 One prompt for everyone, and why it failed

Of all the decisions in this project, this is the one that I took, built and then undid most completely.

My first design of the third block used a single general prompt. That prompt read the document and returned a flat list of data, organised according to the source each value came from. It was the obvious idea and I was confident about it, since one prompt would then serve every appraisal company. It did not work, and finding that out was the sharpest correction I received during the internship.

Three things went wrong at the same time. First, a general prompt does not know where each value lives inside a particular company's template, so it has to search the document instead of being told where to look. Second, the schema had to be written twice, once in the prompt and once as typed models in the code, which made every change to it expensive. Third, data from different sections of the report contaminated each other, because a prompt organised by topic invites the model to gather everything that resembles that topic, wherever it happens to appear.

I abandoned the design when I had seen enough output to accept that the templates were genuinely too different from one another to be covered by a single set of instructions. My tutor and my colleagues already knew this and had told me, and I still had to find it out by trying. I do not consider that time lost, because the three reasons above are now things I understand rather than things I was told, but it is a fair description of how the decision was actually taken.

The design that replaced it uses one prompt per appraisal company, and the extraction visibly improved in the checks I made by hand, although I never measured it. The generality I had been aiming for was not free: it moved work from me to the model, and the model was

worse at that work than I was.

7.6 The schema kept growing

The JSON schema grew while I worked, from about ten sections to eighteen, and the growth came from the control spreadsheet. As I defined the checks of the fourth block, I kept finding values that a check needed and that the schema did not extract, so the schema had to be extended in order to include them. Section 7.12 describes that mechanism, and the growth of the schema is that mechanism working in practice.

Revising the schema also turned up mistakes of my own, such as fields that I had described in the prompt but never included in the structure, and sections that I had written twice.

7.7 Working on the prompt

The prompt is what I spent most time on, by a wide margin, and it is also where I learned the most. The work was trial and error, with many failed attempts, many small adjustments, and a few moments in which I had to start the version again from the beginning.

In an early phase I drafted several prompts at once, one for each of several appraisal companies and a general one, together with a catalogue of the fields to extract and several attempts at the JSON structure. I tried each version on real sample reports and wrote down, version by version, what it got right and what it got wrong. I set that work aside when the design changed, but it was not wasted, because the rules and the field catalogue of the production prompt came out of it.

For the appraisal company that reached production I ended up with five successive versions of the prompt. Inside each version I made continuous small corrections, and I started a new version whenever a change was large enough that patching the previous one no longer made sense. The version in use at the end of the internship is not final either, and it will need further adjustment as more checks are added.

7.7.1 The rule that was hardest to enforce

Each version of the prompt directed the model more precisely. Each one added a set of strict rules, a map of the sections of the report, and an explicit description of the data expected in each of those sections. The rules asked the model to quote the source literally, to take the page number from the page marker, and neither to invent nor to compute anything.

The rule that cost me most effort to enforce was the one that required the model to take each value from a specific *physical section* of the document, rather than from wherever in the document that value happens to appear. The reason why this matters is particular to appraisal reports.

In an appraisal report the same fact is deliberately repeated in several places. The surface area appears in the description written by the appraiser, and again in the land registry note, and again in the cadastral record. That redundancy is the whole point of the document, because the review consists precisely of comparing those copies against each other. Now imagine that a check has to compare the surface stated in the report against the surface stated in the registry note, and that the model, having been asked by topic, has taken both values from the registry note. The check then compares one value against itself. It passes, and it would have passed even if the report and the registry had genuinely disagreed.

That situation is a false pass in the sense described in Section 6.2. The system reports that everything is in order and nothing in its output looks wrong, because the extraction is faithful, the quotation is real and the check does run. The only thing missing is the comparison itself. Getting the model to respect the physical structure of the document was therefore not a question of tidiness, but the condition for the checks to mean anything at all.

7.7.2 Not asking the model to count

An early version of the prompt asked the model how many photographs the report contained. I replaced that question with a list of the photograph captions, copied literally, and let the fourth block count the entries of that list.

The reasoning goes beyond photographs, and it is what lies behind the rule that the model never calculates anything. A number produced by the model is a decision that the model has taken, and it cannot be checked against the document: if the model answers twelve, nothing in the report says twelve. A list of captions is not a decision but an extraction, so every entry can be checked against the text, and counting the entries afterwards takes one line of code that cannot go wrong. Given the choice between a value that the model computes and a value that the model copies, the copy is always preferable, because it keeps every judgement on the side of the system that can be audited.

7.7.3 Why the first real call worked

The company's API key was not available to me during most of the development, so I could not call the model from the pipeline while I was writing the prompt. That did not stop me from

refining it, because I did the work by hand instead, in Microsoft Copilot (Microsoft, 2026), the assistant that the company made available to us, which runs on a GPT model from OpenAI.

The procedure was always the same. I opened a conversation, pasted the system prompt into it, attached the Markdown version of the report, and asked for the JSON. When the assistant answered, I copied the JSON out and checked it against the document value by value: whether each quotation really appeared in the report, whether the page numbers were right, and whether each value had been taken from the section it was supposed to come from. Everything that came back wrong turned into a correction to the prompt, and the next version went through the same procedure again.

That is the reason why the first run against the real service produced a correct and well-formed extraction. It was not a lucky result and I did not experience it as a surprise, because by that point the prompt had already been through many rounds of that manual checking. What the first real call actually tested was the connection between the parts of the system, rather than the prompt itself. Small adjustments continued afterwards, now against the real service, but by then the expensive part of the iteration had already been done, and it had cost nothing.

7.8 Writing the schema in one place only

In my first design the structure of the extracted data was written twice: once in the prompt, in ordinary language, and once as typed models in the code. Every change to the schema then had to be made in two places, and more than once I discovered that I had made it in only one of them.

In the current design the schema lives only in the prompt, and the JSON that the model returns is stored exactly as it arrives. There is a real cost to this decision, because the code no longer enforces the shape of what arrives, and a value of the wrong type is only noticed later, by the block that reads it. What the block does keep is a light check that costs almost nothing: it compares the sections contained in the answer against the sections that the prompt asked for, and it writes down any that are missing. The check does not reject anything, but it does mean that an incomplete answer leaves a visible trace.

I judged that exchange acceptable because the alternative fails in a worse way. A duplicated schema does not fail loudly. It drifts slowly instead, and the two copies end up disagreeing without anyone noticing.

7.9 The shortest version of the call

The single change that taught me most was not an addition but a deletion. My first version of the call to the language model was elaborate, with layers that prepared the request, handled the answer, and covered cases that in the end never occurred. Near the end of the internship my tutor showed me a much shorter way of doing exactly the same thing: send the prompt and the document, ask for JSON, and read the answer. His version fitted in a few lines and removed a considerable amount of code from the block.

What I had done is a common mistake. I had written for the system that I imagined rather than for the system that existed. The elaborate version was not wrong in itself, it simply solved problems that this project did not have. The shorter version is easier to change, and easier for whoever inherits the project to understand. I would not have arrived at it on my own, and seeing the two versions side by side taught me more than writing the first one had.

7.10 The verdict I took out

There is one thing that the first version of the system did and that the final one deliberately does not do. It is the decision that I find hardest to defend in either direction.

The monolithic version ended by producing a single verdict. It gave each detected problem a severity, combined those severities into a score between zero and one, and used that score to label the report either as approved or as needing human review. The result looked finished, because a reviewer could open it and see an answer.

I removed that verdict, and the reason is that the weights behind it were mine and I had no basis for them. Deciding that a missing photograph counts less than a mismatched surface area, and deciding by exactly how much, is a business judgement about how costly each kind of defect is to the company, and nobody had made that judgement. The number that came out of the calculation looked objective without being objective, which is the same objection that I raise against unverifiable numbers in Chapter 6.

The current system therefore reports each check separately and leaves the overall verdict empty. That is worse for whoever wants a single answer and better for whoever has to defend one, and it is the kind of decision that I would rather the company took for itself than have me take on its behalf.

7.11 The local model, which was really a cost question

I looked into running the language model on a local machine and then set the idea aside. The motivation was not curiosity. The price of commercial language models is a recurring external cost that grows with the number of documents reviewed, and it is a cost that the company does not control. In the weekly meetings it was raised that, in the coming years, moving this kind of workload to the company's own servers might become cheaper than paying per call, and that buying a machine for that purpose was being considered. The question is the familiar one of renting against owning. Paying per use avoids both the investment and the maintenance, but it leaves the company exposed to the supplier's prices, and the volume of an automated review system only grows over time.

I could not settle that question. I did prepare the system so that the choice remains open, because the block already treats the provider as a setting in the configuration. Finishing the local path would still be work, but it would not mean rewriting the block. What I did not prepare was the accounting. The block recorded the tokens of every call until the simplification of Section 7.9 removed that record, so the figures the comparison would need are not being collected today.

The attempt never reached the point of running anything. What I had available was an ordinary laptop, with a limited graphics card and limited memory, and models of this kind are not designed to run on a computer of that capacity. The ones I wanted to try were large enough that I could not even finish downloading them onto it, so no report was ever put through a local model.

What that leaves is a question I could not answer rather than a negative result. The comparison that matters would need hardware bought for the purpose, and for that reason the local option stays open for the company rather than closed.

7.12 From a spreadsheet to a contract

One of the more useful things I did in this project was to turn a business document into a technical one.

The company gave me a spreadsheet with a hundred controls spread over thirty-two groups, written by and for the people who review appraisals. It had been produced a year earlier, in a conversation with the reviewing department, as a provisional version that still needed revising and refining. My work with it was to go through it control by control, to keep the ones that made sense to automate, and to ask about the ones whose intent was not

clear. In those cases I needed to know what exactly a given check was looking for, and what would count as passing or failing it.

That spreadsheet then decided which values the model had to extract, because the fourth block can only check a value that the third block has already extracted. In that sense it became the contract between the two halves of the system, extraction and validation, and every extension of the schema described in Section 7.6 can be traced back to a line in it.

On my last day I presented the results that the system produced on real reports to the department that had written that spreadsheet, so that they could carry on from there. That was the intended end of my part of the work. What I left behind was not a finished product, but a working base and a description of it precise enough for someone else to continue.

7.13 Where I got stuck

Two parts of the project delayed me for longer than the rest, and each of them delayed me for a different reason.

The first was the third block. I did not understand well enough how an API call works, nor how to fit such a call into a design organised around classes, and I was learning both things at the same time as I was using them. Section 7.14 returns to what that produced.

The second was the planning itself. I redid the flow diagrams and the Jira board several times, and the reason was not that the tools were difficult but that the design underneath them kept changing. Every reversal reported in this chapter meant a board that no longer described the project. I kept updating it rather than letting it fall out of date, which was the right choice, but it cost time that I had not budgeted for, and the result was never as clean as it should have been.

7.14 What I would do differently

If I started the same project again, with what I know now, I would change four things and keep two.

I would go straight to independent blocks. The monolithic version was not useless, since it proved that the phases worked, but I now know what the final structure looks like and there is no reason to arrive at it through a detour.

I would write one prompt per appraisal company from the first day. This is the same conclusion as Section 7.5, and it is the change that would have saved the most time.

I would look at the available tools more widely, and earlier. I settled on the first reasonable option more often than I should have, both for reading PDFs and for running models. In particular I would try the local option early rather than late, while there is still time to act on what it shows.

I would manage the project properly in Jira. This is the weakness that I am most aware of. With the design changing as often as it did, the board became something that I updated after the fact rather than something that I planned with, and the record it left is poorer than the work it describes. I understand now that this is not administrative overhead. In a company, a board that reflects reality is how other people know what is happening, and it is how the work survives the person who did it. It is a professional habit that I still have to build.

7.14.1 What I would keep

The first thing I would keep is the first block. Identifying each document by the hash of its content, copying it into its own working folder, and doing so in a way that cannot leave a half-written copy behind, is the part of the system that I am most confident about. It is simple, it never lost information, and everything downstream depends on it being right.

The second is the design of the block states. The fact that a block reports what happened rather than merely succeeding or failing, as described in Section 3.3, has held up under every change since, including the move to a chain and the prospect of moving to the cloud.

7.14.2 What I built without understanding it

There is one part of the system that I would not simply improve but redo differently, and honesty requires saying why.

The structure of the third block, with its three interchangeable ways of obtaining an answer, was something I built before I properly understood the pieces it was made of. I did not yet know how the cloud API worked, what a stub was for, or what running a model locally would actually involve. The result was a reasonable-looking design assembled out of concepts that I had not yet used in practice, and its weaknesses were visible once the block had to do real work. When my tutor gave me the short version of the cloud call described in Section 7.9, I integrated his code, removed my own, and left the local and stub paths where they were.

I take two things from this. The first is that a design built out of concepts one does not yet understand tends to be more complicated than the problem rather than less, because complexity is what uncertainty looks like in code. The second is that the right response, once the shape of the problem is clear, is to delete rather than to defend. The block is better for having had my version taken out of it.

Chapter 8
Conclusions

This chapter reviews what the project achieved against the objectives set in Chapter 1. It then assesses the strengths and the limitations of the design, it sets out what I learned from building the system, and it closes with a personal note.

8.1 Achievements with respect to the objectives

The project delivered what it set out to deliver, which was a working pre-production pipeline of four blocks, built from an empty folder within the period of the internship.

Against the objectives that the company set, the outcome is the following. The architecture is modular, it is based on independent blocks, and it is prepared for deployment on the cloud. The first block ingests a PDF, identifies it by the hash of its content and prepares it for the rest of the pipeline. The second block turns the document into text that the model can read. The third block uses a language model to produce structured data in which every value carries a literal quotation as its evidence. The fourth block applies the business rules as deterministic checks written in code, with a catalogue that reached 43 checks in its first complete version and that kept growing afterwards. An orchestrator runs the four blocks as a chain, in the same shape that the system will have once it is deployed on the cloud.

Against the decisions that I contributed myself, which are listed in Section 1.4, all four of them are present in the delivered system. They are the page-by-page conversion that makes every page identifiable, the requirement that every value be either quotable or left empty, the contract of states between the blocks, and a data schema that exists in exactly one place.

Several objectives were not reached, and Section 1.4 lists them. They are extending the system beyond one appraisal company, completing the catalogue of checks, deploying on Azure, finishing the local-model path, and measuring the quality of the extraction. Two of them were agreed to be outside my work from the start, because my tutor decided that the system would

take one appraisal company at a time, and because the deployment on Azure was never part of the internship. Of the remaining three, Chapter 9 sets out what the local-model path and the measurement of quality would require. The third, completing the catalogue of checks, depended on the definitive rules that the company was still writing when the internship ended.

8.2 An honest assessment of the design

There are three strengths of the design that I would defend.

The first is that every value carries its own evidence, so any conclusion that the system reaches can be traced back to a literal fragment of the document and a page number, without anyone having to open the PDF. The second is that the language model never decides whether a report is correct, which keeps every outcome deterministic, reproducible and explicable to a client. The third is that the blocks are independent, which is why I was able to redesign one of them, more than once, without touching the others.

The limits of the design are equally clear, and the first of them qualifies everything that I have just said, because none of those three strengths has been measured. I do not know the precision of the extraction, and I do not know how often the model prefers an empty field to a plausible guess, even though that behaviour is the one on which the entire design depends. Chapter 6 specifies how those figures should be obtained, but it does not stand in for them, and no claim in this thesis should be read as if it did.

Beyond that limitation there are four more. The system has been exercised on the reports of a single appraisal company, so how well the approach transfers to another one remains untested. Some of the checks need data that is not in the PDF at all and would require external sources. No single global verdict is produced, because deciding which failures make a whole report unacceptable is a business rule that was still being written when the internship ended. And two of the three paths of the third block never came into real use, since the local one was left unfinished and the stub was written but never became part of how I worked. That part of the block, discussed in Section 7.14, is the one I would rebuild rather than extend.

8.3 What I learned

This was the first time that I had built a system of this kind, and most of what I learned came from the things that did not work the first time.

Object-oriented programming, in practice rather than in principle. The master's degree taught me what a class is. It did not teach me, because it could not, how to decide what the classes of a real system should be, or when a piece of code should become one and when it should not. I had to build that judgement by making the wrong choice and then living with the consequences, which is the substance of Chapter 7.

How to write for a language model. The prompt is what I spent most time on and what I learned most from, because it taught me a way of thinking that I did not have before. The useful question turned out to be not what the model can do, but what the model should not be allowed to do. Almost every rule that I ended up writing is a restriction, and the system is more trustworthy for each one of them.

That cost is a design constraint. Working with a service that is billed per call changed the way I designed. Deciding in code whatever can be decided in code, before the expensive step is reached, is not an optimisation that gets added at the end. It is a shape that the system has from its first diagram, and it is probably the part of the project that belongs most to this master's degree rather than to a purely technical one.

That simpler is a skill, not a shortcut. My instinct, when I faced something that I did not understand, was to build more, and Section 7.14 describes where that instinct led me. Watching my elaborate version of a component be replaced by a few lines that did the same thing taught me more than writing it had.

How to work when the specification moves. Writing a short specification before each block, and keeping a record of what changed and why, is what allowed a project that was redesigned several times to remain comprehensible. The weakness of my own practice here is the one that I identify in Section 7.14, since I kept that record after the fact rather than planning with it, and that is the professional habit that I most need to build.

8.4 A personal note

Ever since these tools appeared I had wanted to learn how to build automation with them, so when my tutor offered me this project I was genuinely enthusiastic about it. Now that I have finished, that has not changed. This is the kind of work I would like to do: systems that take

a repetitive task off the desk of the people who do it, and then give it back to them in a form that they can check.

The internship also showed me the extent of what I do not yet know, and where the boundary lies between using a tool well and understanding what it does. That boundary is where I intend to keep working.

Chapter 9
Future Work

This chapter lists the work that was left undone. Most of it was out of reach during the internship, either because there was not enough time or because the code stays with the company.

9.1 Running the evaluation

The most important next step is to carry out the protocol described in Chapter 6, because running it would turn this descriptive work into a measured one.

Its three levels should be attempted in order. The first level is the automatic check of the literal quotations, which needs no annotation and could be run immediately. The second level is the field-by-field accuracy of the extraction, measured against a set of reports that a reviewer has annotated by hand. The third level is the agreement between the system and a human reviewer over the whole catalogue of checks, and it has to wait until the company fixes its definitive rules.

Before all three, the token accounting that was removed from the third block should be put back, because without it there is no cost per document to report.

9.2 Additional appraisal companies

The system supports one appraisal company so far, and adding another one does not require any change in the rest of the code. It is enough to study the reports of the new company, write its prompt, and add the clues that let the third block recognise its reports. This is the most obvious next step, and the one that would show whether the approach really transfers from one template to another.

9.3 Reading the scanned annexes

Some annexes arrive as scanned images and cannot be read as text at all. Two improvements would help, each at a different point of the pipeline.

The first improvement is to decide, before anything is extracted, whether a report needs optical character recognition, using a classifier of the kind described in Section 2.4.1. Running such a classifier over the project's own reports would be cheap, and it would measure how common the mixed case really is.

The second improvement is to read the scans themselves, which is harder than it sounds for a document of this length. Ordinary OCR tools and vision models tend to run out of memory when they have to keep many pages in view at the same time.

One recent model addresses exactly that problem. *Unlimited OCR* (Yin et al., 2026), published by Baidu, changes the way the model remembers what it has already read: instead of letting that memory grow with every page, it keeps the memory at a fixed size. The practical effect is that the model can read tens of pages in a single pass rather than page by page, and it does not slow down as the document gets longer. Whether it reads Spanish appraisal annexes well enough would have to be tried on the company's own documents.

9.4 Running the model on the company's own hardware

Paying a provider per call is a cost that grows with the number of documents reviewed and that the company does not control, whereas buying a machine replaces that cost with an investment that the company owns. I looked into the local option during the project and then set it aside because of the limits of the laptop that I had available, as explained in Chapter 7.

Turning the question into a decision needs two things that I could not obtain. The first is adequate hardware, meaning a machine bought for the purpose that can hold both the model and a complete report in memory at the same time. The second is the comparison itself, which starts by restoring the token accounting. Running the same documents locally would then give the throughput that has to be set against the price of the machine.

Until someone calculates at what volume of reports the machine pays for itself, the choice between renting and owning remains an opinion rather than an answer.

9.5 Cloud deployment on Azure

The blocks are ready to be deployed on Microsoft Azure, where the arrival of a document would start the pipeline automatically. That deployment, together with the orchestration between the blocks, is the next step towards a production system.

9.6 Whether the per-block states are still needed

Once the pipeline runs unattended on the cloud, most documents should travel through it in a single pass and leave nothing half-done. The only “already processed” result that would normally appear is the deduplication carried out by the first block, so the per-block states could start to look redundant.

The argument for keeping them is that the exceptional cases are the expensive ones. A document may stop halfway through the chain and be sent again, or the orchestration may retry a step after a temporary failure. In both of those cases the per-block states are what prevents the language model from being called a second time for a document that has already been extracted. Removing them would make the pipeline simpler and, in those particular cases, more expensive.

Taking that decision needs figures on how often documents are re-run in production. I did not have those figures, so the decision is left to the company.

9.7 Integration with external data sources

Some checks need data that is not in the appraisal report at all, such as the data held by the public cadastre or by the national statistics office. Connecting the fourth block to those sources would complete the set of checks that cannot be carried out with the PDF alone.

References

Anthropic. (2026). *Claude.* [Large language model]. (https://claude.ai/)

Arner, D. W., Barberis, J., & Buckley, R. P. (2017). FinTech, RegTech, and the reconceptualization of financial regulation. *Northwestern Journal of International Law & Business*, *37*(3), 371–413.

Banco de España. (2017). *Circular 4/2017, de 27 de noviembre, del banco de españa, a entidades de crédito, sobre normas de información financiera pública y reservada, y modelos de estados financieros.* Boletín Oficial del Estado, de 6 de diciembre de 2017. BOE-A-2017-14334. (Spain)

Brown, T. B., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. In *Advances in neural information processing systems* (Vol. 33, pp. 1877–1901).

European Parliament and Council of the European Union. (2024). *Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act).* Official Journal of the European Union, 12 July 2024. (http://data.europa.eu/eli/reg/2024/1689/oj)

Firecrawl. (2026). *pdf-inspector: a Rust library for PDF classification and text extraction.* [Computer software]. MIT License. (https://github.com/firecrawl/pdf-inspector)

Gao, T., Yen, H., Yu, J., & Chen, D. (2023). Enabling large language models to generate text with citations. In *Proceedings of the 2023 conference on empirical methods in natural language processing (EMNLP)* (pp. 6465–6488). Singapore.

Geng, S., Josifoski, M., Peyrard, M., & West, R. (2023). Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 conference on empirical methods in natural language processing (EMNLP)* (pp. 10932–10952). Singapore.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., … Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, *55*(12). doi: 10.1145/3571730

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., … Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in neural information processing systems* (Vol. 33, pp. 9459–9474).

Li, Z., Abulaiti, A., Lu, Y., Chen, X., Zheng, J., Lin, H., … Sun, L. (2024). *READoc: A unified benchmark for realistic document structured extraction.* Retrieved from https://arxiv.org/abs/2409.05137

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association*

for Computational Linguistics, *12*, 157–173. doi: 10.1162/tacl_a_00638

Livathinos, N., Auer, C., Lysak, M., Nassar, A., Dolfi, M., Vagenas, P., … Staar, P. W. J. (2025). *Docling: An efficient open-source toolkit for AI-driven document conversion.* Retrieved from `https://arxiv.org/abs/2501.17887`

Micheler, E., & Whaley, A. R. (2020). Regulatory technology: Replacing law with computer code. *European Business Organization Law Review*, *21*(2), 349–377. doi: 10.1007/s40804-019-00151-1

Microsoft. (2026). *Microsoft Copilot.* [Large language model]. (`https://copilot.microsoft.com/`)

Ministerio de Economía. (2003). *Orden ECO/805/2003, de 27 de marzo, sobre normas de valoración de bienes inmuebles y de determinados derechos para ciertas finalidades financieras.* Boletín Oficial del Estado, núm. 85, de 9 de abril de 2003, pp. 13678–13707. BOE-A-2003-7253. (Spain. ELI: https://www.boe.es/eli/es/o/2003/03/27/eco805)

Ministerio de Economía y Hacienda. (2009). *Real decreto 716/2009, de 24 de abril, por el que se desarrollan determinados aspectos de la ley 2/1981, de 25 de marzo, de regulación del mercado hipotecario y otras normas del sistema hipotecario y financiero.* Boletín Oficial del Estado, núm. 107, de 2 de mayo de 2009, pp. 38490–38516. BOE-A-2009-7352. (Spain)

Ouyang, L., Qu, Y., Zhou, H., Zhu, J., Zhang, R., Lin, Q., … He, C. (2025). OmniDocBench: Benchmarking diverse PDF document parsing with comprehensive annotations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (CVPR)* (pp. 24838–24848).

Rashkin, H., Nikolaev, V., Lamm, M., Aroyo, L., Collins, M., Das, D., … Reitter, D. (2023). Measuring attribution in natural language generation models. *Computational Linguistics*, *49*(4), 777–840.

Smith, R. (2007). An overview of the Tesseract OCR engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)* (pp. 629–633).

Stang, M., Krämer, B., Nagl, C., & Schäfers, W. (2022). From human business to machine learning—methods for automating real estate appraisals and their practical implications. *Zeitschrift für Immobilienökonomie*. doi: 10.1365/s41056-022-00063-1

Xu, D., Chen, W., Peng, W., Zhang, C., Xu, T., Zhao, X., … Chen, E. (2024). Large language models for generative information extraction: A survey. *Frontiers of Computer Science*, *18*(6). doi: 10.1007/s11704-024-40555-y

Xu, Y., Li, M., Cui, L., Huang, S., Wei, F., & Zhou, M. (2020). LayoutLM: Pre-training of text and layout for document image understanding. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery and data mining (KDD '20)* (pp. 1192–

1200). doi: 10.1145/3394486.3403172

Yin, Y., Liu, H., YY, Xie, Q., Liu, C., Yang, S., … Jia, L. (2026). *Unlimited ocr works.* Retrieved from https://arxiv.org/abs/2606.23050

APPENDICES

Appendix A

Project Planning Detail

The Jira board used during the project is no longer available, because it was removed after the end of the internship. This appendix reproduces the planning from the project's own documents. The same structure is summarised in Chapter 4.

The project was organised as one EPIC, "a container pipeline for the automatic review of appraisal reports under Order ECO/805/2003", divided into the user stories of Table A.1 and the tasks of Table A.2.

Table A.1: User stories.

Story	Title	Status
H0	Overview of the pipeline (transversal)	Reference
H1	Ingestion and deduplication	Finished
H2	Text and table extraction	Finished
H3	Structured extraction with a language model	Finished (redesigned)
H4	Validation and result	First version implemented

Table A.2: Task breakdown by story.

ID	Task
T1.1	Hashing and filesystem utilities
T1.2	Ingestion result model and states
T1.3	Configuration
T1.4	Ingestion service: validate, hash, deduplicate and copy

ID	Task
T1.5	Cleanup of the working folder by age
T1.6	Command-line entry point
T1.7	Optional deletion of the input PDF after processing
T1.8	Optional request file copied together with the PDF
T2.1	Evidence and page models
T2.2	Extraction result model and states
T2.3	Configuration
T2.4	Plain-text extractor with inline tables
T2.5	Extraction service with two outputs (non-blocking Markdown)
T2.6	Command-line entry point
T2.7	Markdown extractor
T3.1	Model-client interface and factory
T3.2	Cloud, local and stub providers
T3.3	Appraiser detection (cascade, without the model)
T3.4	Appraiser registry and prompt loader
T3.5	JSON parser that tolerates a messy answer
T3.6	Extraction service
T3.7	Command-line entry point
T4.1	Result models and states
T4.2	Defensive reading and value normalisation
T4.3	Control catalogue (18 groups, 43 checks)
T4.4	Validation service engine
T4.5	Configuration (thresholds)
T4.6	Command-line entry point

Appendix B

Extracted Data Schema (18 blocks)

The structured data produced by the third block of the pipeline is organised into eighteen blocks, one for each main part of the report. The table below describes each block at a conceptual level. The exact instructions given to the model are part of the prompt, which is not reproduced.

Table B.1: The eighteen blocks of the extracted data schema.

Block	Content
Identification	Applicant, document type, address, file number and date.
Index and annexes	The report index and which annexes are present.
Applicant and purpose	The regulatory purpose of the appraisal and the type of visit.
Identification and location	Property type, registry data, cadastral reference, address and coordinates.
Checks and documentation	The list of checks performed by the appraiser.
Land description and surface	The surface of the plot.
Building description and surface	Age, condition, equipment and energy rating.
Urban-planning description	The planning situation of the property.
Protection, tenure and occupation	Protected-housing status, occupation and owners.
Technical values and calculation	Surfaces, methods applied, value of the land and insurance values.

Block	Content
Comparison method	The comparable properties used and their adjustments.
Income method	Data for the income method (not applicable to ordinary dwellings).
Cost method	Replacement-cost data.
Appraisal value	The final value, the mortgage value and the relevant dates.
Photographs	The list of photo captions found in the report.
Plans	Which location and site plans are present.
Registry note	Data from the land registry note: owners, charges and surfaces.
Cadastral record	Data from the cadastral document: reference, year, surface and use.

Appendix C

Validation Control Categories

The fourth block runs a catalogue of checks, grouped by category. The first version has 43 checks in the eighteen groups below. The catalogue kept growing afterwards, as explained in Chapter 5. Each check returns one of four outcomes: pass, fail, no data, or not applicable.

Table C.1: The eighteen groups of the validation catalogue.

Group	What it checks
Purpose	The regulatory purpose of the appraisal.
Documentation	That the required documents are present.
Registry note	Consistency with the land registry note.
Dwelling type	The type of property.
Registry data	The registry identifiers of the property.
Cadastral reference	The cadastral reference and its format.
Registry identifier (IDUFIR)	The unique identifier of the registered property.
Protected housing	Whether the property is protected housing.
Visit	The type and date of the visit.
Photographs	The photographs included in the report.
Plans and sketch	The presence of the location and site plans and the sketch.
Owners	That the listed owners are consistent across documents.
Age	The age of the building.
Surfaces	That the surfaces agree across the report, the registry note and the cadastre.

Group	What it checks
Values	The assigned value and the mortgage value.
Checks	The checks declared by the appraiser.
Conditions and warnings	The conditions and warnings stated in the report.
Cost method	The data of the cost method.

Appendix D

Domain Glossary

Table D.1: Glossary of domain terms.

Term	Definition
Order ECO/805/2003	The Spanish ministerial order that regulates the valuation of property for financial purposes.
Appraisal company	A valuation company approved by the Bank of Spain to issue official appraisals.
Mortgage-guarantee purpose	The valuation purpose used when the property is the collateral of a mortgage.
Mortgage value	The value of the property for the purpose of the guarantee.
Registry note (nota simple)	An informative certificate from the Land Registry, with the owners, charges and surfaces.
Cadastral reference	The identifier of a property in the Cadastre.
Cadastral record	The Cadastre document with the surface, use and year of construction.
IDUFIR	The unique identifier of a registered property.
Protected housing	Housing under a public protection scheme, with a legal maximum value.
Comparable	A similar property used in the comparison method.
Evidence	The pair (literal quotation, page) that justifies an extracted value.

Term	Definition
Idempotency	The property of running a step again without repeating its effects.
SHA256	The content hash used as the identifier of each document.
Stub	A fake implementation of the model, used to run the third block offline and at no cost.
System prompt	The set of instructions that guides the language model.

Appendix E

Per-Block States Reference

Every block ends by returning a *state*. The state is the only thing the next step of the pipeline needs to read, and it is written to the block's result file so that the outcome of a document can be inspected afterwards. The states keep their original Spanish names from the code.

The states of the four blocks fall into three families, described in Section 3.3: states that let the document continue, states that stop it for an expected reason, and states that stop it because something went wrong. Table E.1 lists them.

Table E.1: States of the four blocks, grouped by what they mean for the chain.

Block	Continue	Stop, expected	Stop, error
Block 01	INGESTADO, YA_EXISTE	(none)	ERROR_NO_EXISTE, ERROR_NO_PDF, ERROR_NOMBRE_INVALIDO
Block 02	PROCESADO, YA_PROCESADO	PDF_ILEGIBLE	ERROR_PDF, ERROR_SHA_NO_EXISTE, ERROR_SIN_PDF
Block 03	PROCESADO, YA_PROCESADO	TASADORA_NOLISTA	ERROR_SIN_FUENTES, ERROR_LLM
Block 04	VALIDADO, YA_VALIDADO	(none)	ERROR_SIN_EXTRACCION, ERROR_DATOS

Two states sit in the middle column. `PDF_ILEGIBLE` is a report that arrives as a scan, and `TASADORA_NOLISTA` is a report from an appraisal company with no prompt written yet. Neither is a failure of the system, and Chapter 5 explains why each one stops the document.

Appendix F

Implementation Pseudocodes

This appendix provides the pseudocode for the four blocks of the REVIN pipeline and for the orchestrator that runs them as a chain.

F.1 Block 01: Ingestion and Deduplication

```
# ================================================================
# BLOCK 01 - INGESTION
# Move a PDF from INPUT/ into its own folder WORK/{sha256}/,
# deduplicated by the SHA256 of its content.
# ================================================================

# ---- The states this block can end in -------------------------
# (the names are the ones used in the code, in Spanish)
ENUM IngestionState:
    INGESTED                # "INGESTADO"  → new PDF, copied into WORK/
    ALREADY_EXISTS          # "YA_EXISTE"  → same PDF seen before;
    ↪ nothing is copied
    ERROR_NOT_FOUND         # "ERROR_NO_EXISTE"
    ERROR_NOT_PDF           # "ERROR_NO_PDF"
    ERROR_INVALID_NAME      # "ERROR_NOMBRE_INVALIDO"

# The first two let the document continue; the three errors stop it.
CONSTANT CONTINUES = { INGESTED, ALREADY_EXISTS }

# ---- What the block reports back ------------------------------
```

```
CLASS IngestionResult:          # saved as ingesta_result.json inside the folder
    state          : IngestionState
    original_name  : text                  # the PDF keeps its name inside WORK/
    sha256         : text OR NULL          # the identifier of the document
    size_bytes     : integer OR NULL
    ingestion_date : datetime
    request_found  : boolean = FALSE       # an optional request may travel with the PDF
    request_valid  : boolean OR NULL       # NULL when there is no request

# ---- Configuration ------------------------------------------------
# Read from the environment, with defaults. Everything that could change
# when moving to the cloud lives here and nowhere else.
CLASS Settings:
    INPUT_DIR = "INPUT"  ;  WORK_DIR = "WORK"
    DELETE_INPUT_AFTER = FALSE       # remove the source PDF once it is safely copied
    CLEAN_ON_INGEST    = FALSE       # run the cleanup below after each ingestion
    RETENTION_DAYS     = 14          # working folders older than this may be removed

# ---- Entry point --------------------------------------------------
# The block RETURNS ITS STATE; there is no exit code. The orchestrator reads
# the state and decides whether the chain goes on (see the state contract).
METHOD main(file_name):
    result ← NEW IngestionService(NEW Settings()).ingest(file_name)
    PRINT result AS json            # work record, also saved to disk
    RETURN result.state

# ---- The service --------------------------------------------------
```

```
CLASS IngestionService:
    settings : Settings

    METHOD ingest(file_name) RETURNS IngestionResult:
        # 1) Cheap checks first, before touching or copying anything.
        IF NOT is_safe_name(file_name):        # a plain file name,
    ↪ never a path
            RETURN result(ERROR_INVALID_NAME)
        source ← settings.INPUT_DIR / file_name
        IF NOT source.is_file():
            RETURN result(ERROR_NOT_FOUND)
        IF NOT is_pdf(source):                 # extension AND "%PDF"
    ↪ header
            RETURN result(ERROR_NOT_PDF)

        # 2) The identifier of the document is the hash of its content.
        sha256 ← compute_sha256(source)
        folder ← settings.WORK_DIR / sha256

        # 3) Deduplication: if the folder exists, this PDF was already
    ↪ ingested.
        #    Nothing is copied again, and this is NOT an error. The
    ↪ source PDF may
        #    still be removed, since the content is already safe in WORK.
        IF folder.exists():
            IF settings.DELETE_INPUT_AFTER:  delete(source)
            RETURN result(ALREADY_EXISTS, sha256, request already in the
    ↪ folder)

        # 4) New document: create its folder and copy the PDF, keeping
    ↪ its name.
        #    The copy is written to a temporary file and then renamed,
    ↪ so a folder
        #    never holds a half-copied PDF.
        create(folder)
        copy_safely(source, folder / file_name)

```

```
        # 5) A request file may travel with the PDF (same name,
    ↪ ".json"). It is
        #    optional: if it is missing, that is not an error; the block
    ↪ just
        #    records that there was none. Block 04 will use it if it is
    ↪ there.
        (found, valid) ← copy_request_if_present(file_name, folder)

        # 6) Write the result, so the state of the document can be read
    ↪ later.
        result ← NEW IngestionResult(INGESTED, file_name, sha256, ...)
        write_json(folder / "ingesta_result.json", result)

        # 7) Optional housekeeping: remove working folders older than
    ↪ RETENTION_DAYS.
        IF settings.CLEAN_ON_INGEST:
            CleanupService(settings).clean()

        # 8) Only now, with the copy confirmed, may the source PDF be
    ↪ removed.
        IF settings.DELETE_INPUT_AFTER AND folder contains the PDF:
            delete(source)

        RETURN result

# ---- The two checks that matter --------------------------------
FUNCTION compute_sha256(path) RETURNS text:
    # Read the file in chunks and feed them to the hash, so a large PDF
    ↪ does
    # not have to be held in memory. The result is deterministic: the
    ↪ same
    # content always gives the same identifier.
    ...

FUNCTION is_pdf(path) RETURNS boolean:
    # Not enough to trust the ".pdf" extension: also read the first
    ↪ bytes and
```

```
    # check that they are "%PDF". This catches renamed or corrupt files.
    ...
```

Pseudocode F.1: Block 01, ingestion and deduplication

F.2 Block 02: PDF Text Extraction

```
# ================================================================
# BLOCK 02 - PDF TEXT EXTRACTION
# From the PDF in WORK/{sha256}/ produce two views of the document:
# a plain text one and a Markdown one. Block 03 reads the Markdown.
# ================================================================

# ---- The states this block can end in -------------------------
ENUM ExtractionState:
    PROCESSED           # "PROCESADO"     → at least one view produced
    ↪ text
    ALREADY_PROCESSED   # "YA_PROCESADO"  → done before; the work is
    ↪ not repeated
    PDF_UNREADABLE      # "PDF_ILEGIBLE"  → the PDF opened but has no
    ↪ text (a scan)
    ERROR_PDF           # "ERROR_PDF"     → the PDF could not be read
    ↪ at all
    ERROR_SHA_NOT_FOUND # "ERROR_SHA_NO_EXISTE"
    ERROR_NO_PDF        # "ERROR_SIN_PDF"

CONSTANT CONTINUES = { PROCESSED, ALREADY_PROCESSED }
# PDF_UNREADABLE stops the document for an EXPECTED reason: the report
    ↪ arrived
# as a scanned image and there is no OCR yet. It is reported, not
    ↪ treated as a bug.

# ---- What the block reports back ------------------------------
CLASS ExtractionResult:         # saved as extraccion_result.json
    state           : ExtractionState
    sha256          : text
    num_pages       : integer OR NULL
```

```
    text_coverage    : number       # share of pages with text, 1.0 =
    ↪ all, 0.0 = none
    markdown_coverage: number       # the same, for the Markdown view
    text_error       : text OR NULL # why a branch failed, if it did
    markdown_error   : text OR NULL

# ---- Entry point ------------------------------------------------
METHOD main(sha256):
    result ← NEW ExtractionService(NEW Settings()).extract(sha256)
    PRINT result AS json
    RETURN result.state

# ---- The service ------------------------------------------------
CLASS ExtractionService:
    settings           : Settings
    text_extractor     : PlainTextExtractor   # both are passed in, so
    ↪ either can
    markdown_extractor : MarkdownExtractor    # be replaced by a fake

    METHOD extract(sha256) RETURNS ExtractionResult:
        folder ← settings.WORK_DIR / sha256
        IF NOT folder.is_dir():                        # Block 01 should
        ↪ have created it
            RETURN result(ERROR_SHA_NOT_FOUND)

        # Idempotency: if this document was already extracted, say so
        ↪ and stop.
        IF a previous result exists AND its state was PROCESSED:
            RETURN result(ALREADY_PROCESSED)

        pdf ← locate_pdf(folder)      # by the name recorded by Block 01
        IF pdf IS NULL:
            RETURN result(ERROR_NO_PDF)

        # The two branches are INDEPENDENT and neither blocks the other:
        ↪ a failure
        # in one is recorded and the other one still runs.
```

```
        TRY:   plain ← text_extractor.process(pdf)
        CATCH: plain ← NULL ; text_error ← the error

        TRY:   markdown ← markdown_extractor.process(pdf)
        CATCH: markdown ← NULL ; markdown_error ← the error

        # Decide the state from what the two branches obtained.
        IF plain failed AND markdown produced nothing:
            RETURN result(ERROR_PDF)                    # the PDF could
    ↪ not be read
        IF neither view produced any text:
            RETURN result(PDF_UNREADABLE)               # readable file,
    ↪ but it is a scan

        write whatever each branch produced:
            texto_extraido.txt   (if the plain view has text)
            texto_extraido.md    (if the Markdown view has text)
        RETURN result(PROCESSED)

# ---- The two extractors ----------------------------------------
# Both offer the same method, process(pdf), so the service treats them
    ↪ alike
# and does not need to know which library is behind each one.

CLASS PlainTextExtractor:
    METHOD process(pdf) RETURNS ProcessedDocument:
        # Page by page: take the text, and take the tables separately so
    ↪ that a
        # table is not written twice. Each page is prefixed with a marker
        # "=== Página N ===" so the page of a quotation can be recovered
    ↪ later.
        # Its weakness: the tables end up detached from the text around
    ↪ them.
        ...

CLASS MarkdownExtractor:
    METHOD process(pdf) RETURNS ProcessedDocument:
```

```
        # Converts the document ONE PAGE AT A TIME, for two reasons: it keeps the
        # tables in a form the model reads better, and converting page by page is
        # what allows the same "=== Página N ===" markers to be added.
        ...

FUNCTION coverage(pages) RETURNS number:
    # A rough quality signal: how many pages came out with text. A coverage of
    # 0.0 is what identifies a scanned document.
    RETURN count(pages with text) / count(pages)
```

Pseudocode F.2: Block 02, PDF text extraction

F.3 Block 03: Structured Extraction with a Language Model

```
# ================================================================
# BLOCK 03 - EXTRACTION WITH A LANGUAGE MODEL
# Detect the appraisal firm (no model involved) → load its prompt →
# ask the model for the report as JSON → save that JSON as it comes.
# The model only EXTRACTS. Deciding whether the report is correct is
# the job of Block 04.
# ================================================================

# ---- The states this block can end in -------------------------
ENUM LlmState:
    PROCESSED           # "PROCESADO"        → data extracted and saved
    ALREADY_PROCESSED   # "YA_PROCESADO"     → done before; the model is NOT called again
    FIRM_NOT_LISTED     # "TASADORA_NOLISTA" → no prompt for that firm yet; no model call
    ERROR_NO_SOURCE     # "ERROR_SIN_FUENTES"→ the Markdown from Block 02 is missing
```

```
    ERROR_LLM           # "ERROR_LLM"        → the call failed, or the
    ↪ answer was not JSON

CONSTANT CONTINUES = { PROCESSED, ALREADY_PROCESSED }
# FIRM_NOT_LISTED stops the document for an EXPECTED reason and costs
    ↪ nothing:
# the check happens BEFORE the expensive call, never after.

# ---- What the block reports back ------------------------------
CLASS LlmExtractionResult:      # saved as extraccion_llm_result.json
    state       : LlmState
    sha256      : text
    firm            : text OR NULL       # the firm that was detected
    model           : text OR NULL       # which model answered
    missing_blocks : list of text       # expected sections the answer
    ↪ did not include
    error           : text OR NULL

# ---- Which firms the system knows -----------------------------
# One entry per studied firm. Adding a firm means writing its prompt and
    ↪ adding
# an entry here; no other code changes. Only one firm was studied during
    ↪ the
# internship, so the registry has one entry.
CONSTANT REGISTERED_FIRMS = {
    "firm_a": { prompt_file    : "system_prompt_firmA.md",
                aliases        : [...],    # looked for in the file name
    ↪ / the request
                marks          : [...],    # looked for in the text of
    ↪ the document
                expected_blocks: [...] }   # the 18 section names its
    ↪ prompt asks for
}

# ---- Entry point ----------------------------------------------
METHOD main(sha256):
    model_client ← choose_model_client(NEW Settings())
```

```
    result ← NEW LlmExtractionService(NEW Settings(),
    ↪ model_client).extract(sha256)
    PRINT result AS json
    RETURN result.state

# ---- Where the answer comes from ------------------------------
# Three ways of obtaining the JSON, chosen by configuration. They all
    ↪ offer the
# same method, generate_json(system_prompt, document), so the service
    ↪ does not
# know, and does not care, which one is behind it.
#   - CLOUD : the one actually in use.
#   - STUB  : returns a fixed answer, so the block can run with no API
    ↪ key and at
#             no cost. Written early on, but never really used.
#   - LOCAL : a model running on the company's own machine. Started, not
    ↪ finished.
FUNCTION choose_model_client(settings):
    IF settings.PROVIDER == "cloud":  RETURN NEW CloudClient(settings)
    IF settings.PROVIDER == "stub":   RETURN NEW StubClient()
    IF settings.PROVIDER == "local":  RETURN NEW LocalClient(settings)

CLASS CloudClient:
    METHOD generate_json(system_prompt, document) RETURNS data:
        answer ← call the API with:
            system  = system_prompt        # the instructions written
    ↪ for that firm
            user    = document             # the Markdown, exactly as
    ↪ Block 02 left it
            format  = JSON                 # the API is told to answer
    ↪ with JSON
            temperature = 0                # same input, same output
        RETURN parse_json(answer.content)

# ---- Detecting the firm, without the model --------------------
CLASS FirmDetector:
    METHOD detect(folder, document) RETURNS firm OR NULL:
```

```
        # Three clues, tried from the most reliable to the least, stopping at the
        # first match. All of it is ordinary code: no model is involved, because
        # this decision has to be free.
        1) the original name of the PDF        (recorded by Block 01)
        2) the request file, if there is one
        3) marks that appear in the text of the document itself
        RETURN NULL if none of them matches   # → FIRM_NOT_LISTED

# ---- The service ------------------------------------------------
CLASS LlmExtractionService:
    METHOD extract(sha256) RETURNS LlmExtractionResult:
        folder ← settings.WORK_DIR / sha256

        # 1) Idempotency. Only a previous PROCESSED result is honoured: a document
        #    that stopped as FIRM_NOT_LISTED is retried, so adding a firm later
        #    only requires running it again.
        IF a previous result exists AND its state was PROCESSED:
            RETURN result(ALREADY_PROCESSED)

        # 2) The Markdown from Block 02 is the only thing the model reads.
        IF the Markdown is missing or empty:
            RETURN result(ERROR_NO_SOURCE)

        # 3) Detect the firm BEFORE spending anything.
        firm ← FirmDetector.detect(folder, markdown)
        IF firm IS NULL:
            RETURN result(FIRM_NOT_LISTED)

        # 4) Load the prompt written for that firm and ask the model.
        system_prompt ← read(REGISTERED_FIRMS[firm].prompt_file)
        TRY:
            data ← model_client.generate_json(system_prompt, markdown)
```

```
        CATCH error:
            # The failed attempt is written to disk before returning, so
    ↪ the reason
            # is still there afterwards. Re-running retries it.
            save(result(ERROR_LLM, error))
            RETURN result(ERROR_LLM, error)

        # 5) Light check, INFORMATIVE ONLY: which of the expected blocks
    ↪ did not
        #    come back? Nothing is rejected. The list goes into the work
    ↪ record so
        #    a person can see that the answer was incomplete.
        missing ← [b FOR b IN REGISTERED_FIRMS[firm].expected_blocks
                   WHERE b NOT IN keys(data)]

        # 6) Save the model's JSON EXACTLY AS IT CAME. The shape of the
    ↪ data is
        #    defined in the prompt and nowhere else, so there is a
    ↪ single place
        #    to change when a field is added.
        write_json(folder / "informe_extraido.json", data)
        RETURN result(PROCESSED, firm, missing_blocks=missing)
```

Pseudocode F.3: Block 03, structured extraction with a language model

F.4 Block 04: Validations and Result

```
# ================================================================
# BLOCK 04 - VALIDATIONS  (ordinary code, NO language model)
# Read the JSON from Block 03 and run a catalogue of checks taken
# from the company's control spreadsheet. The model extracted the
# data; this block is the one that judges it.
# Generic: the checks are the same for every appraisal firm.
# ================================================================

# ---- The states this block can end in -------------------------
ENUM ValidationState:
```

```
    VALIDATED            # "VALIDADO"             → the checks ran and
    ↪ were saved
    ALREADY_VALIDATED    # "YA_VALIDADO"          → a result already
    ↪ existed
    ERROR_NO_EXTRACTION # "ERROR_SIN_EXTRACCION" → the Block 03 JSON is
    ↪ missing
    ERROR_DATA           # "ERROR_DATOS"          → that JSON cannot be
    ↪ read

# ---- How ONE check can end ------------------------------------
ENUM CheckState:
    OK                # checked, and correct
    KO                # checked, and wrong
    NO_DATA           # "SIN_DATOS" → the check applied, but a value it
    ↪ needs is missing
    NOT_APPLICABLE    # "NO_APLICA" → the check makes no sense for this
    ↪ document

# The distinction between the last two is deliberate. "The report does
    ↪ not say"
# is not the same as "this does not apply here", and a reviewer needs to
    ↪ tell
# them apart to know whether something is actually missing from the
    ↪ report.

# ---- What a check leaves behind -------------------------------
CLASS CheckResult:
    id, group, name, description : text   # taken from the control
    ↪ spreadsheet
    spreadsheet_row : integer OR NULL     # so a check can be traced to
    ↪ its origin
    state  : CheckState
    reason : text                         # in plain words: why it ended
    ↪ like that
    compared_values : map                 # the values it used, already
    ↪ normalised
```

```
    evidences : list of (field, quotation, page)   # inherited from
    ↪ Block 03

CLASS ValidationResult:                    # saved as
    ↪ validacion_result.json
    state : ValidationState
    sha256 : text
    verdict : text OR NULL                 # RESERVED: see below
    totals, counts_by_group : counts of OK / KO / NO_DATA /
    ↪ NOT_APPLICABLE
    failed : list of check id              # quick access to what went
    ↪ wrong
    checks : list of CheckResult

# ---- The catalogue ------------------------------------------------
# One module per group of the spreadsheet; each exports its list of
    ↪ checks, and
# all of them are joined into a single catalogue. Adding a group means
    ↪ writing
# its module and adding it here. Nothing else changes.
CONSTANT CATALOGUE = purpose.CHECKS + documentation.CHECKS +
    ↪ registry_note.CHECKS
                    + ... + age.CHECKS + areas.CHECKS + values.CHECKS   #
    ↪ 18 groups, 43 checks

# ---- Thresholds ---------------------------------------------------
# Read from configuration, so a criterion can be changed without
    ↪ touching code.
CLASS Settings:
    REPORT_VALIDITY_MONTHS = 6 ; NOTA_SIMPLE_VALIDITY_MONTHS = 3
    AGE_TOLERANCE_YEARS = 0        # decided: the years must match
    ↪ exactly
    AREA_TOLERANCE_PCT  = 0.0      # decided: the areas must match
    ↪ exactly

# ---- Entry point --------------------------------------------------
METHOD main(sha256):
```

```
    result ← NEW ValidationService(NEW Settings()).validate(sha256)
    PRINT summary(result)          # the full detail goes to
    ↪ validacion_result.json
    RETURN result.state

# ---- The service ------------------------------------------------
CLASS ValidationService:
    METHOD validate(sha256) RETURNS ValidationResult:
        folder ← settings.WORK_DIR / sha256

        IF a result already exists:
            RETURN result(ALREADY_VALIDATED)        # deleting the file
    ↪ forces a re-run

        report ← read_json(folder / "informe_extraido.json")
        IF it is missing:            RETURN result(ERROR_NO_EXTRACTION)
        IF it cannot be read:        RETURN result(ERROR_DATA)
        request ← read_json_if_present(folder / "peticion.json")   #
    ↪ optional

        # Run the WHOLE catalogue. A check that breaks must not stop the
    ↪ others:
        # it is recorded as NO_DATA and the block carries on. One faulty
    ↪ rule
        # cannot cost the reviewer the other forty-two results.
        FOR EACH check IN CATALOGUE:
            TRY:    results.add(check.run(report, request, settings))
            CATCH:  results.add(NO_DATA, reason="this check is broken")

        count the states, write validacion_result.json, RETURN VALIDATED

# The global verdict is RESERVED, and left empty on purpose. Deciding
    ↪ which
# failures make a whole report unacceptable is a business rule, not a
    ↪ technical
# one, and the final set of rules was not available during this work.
    ↪ The block
```

```
# reports what passed and what failed; who turns that into a single yes
    ↪ or no is
# a decision for the company.

# ---- Reading the JSON without breaking ------------------------
# The extracted report may be incomplete: whole sections can be empty
    ↪ when an
# annex is missing from the PDF. The readers therefore never fail on
    ↪ absent
# data; they return "nothing", and the check that asked for it ends as
    ↪ NO_DATA.
FUNCTION read_field(report, path) RETURNS (value, evidence):
    # Each field arrives as { value, source_text, source_page }, so
    ↪ reading a
    # value also gives the quotation and the page it came from.
    ...

# ---- Making values comparable ---------------------------------
# The same fact is written differently in the report, the registry note
    ↪ and the
# cadastral record. Before two values can be compared, they must be
    ↪ normalised.
FUNCTION normalise_number(v)  # "150.000,00 Euros" → 150000.00
    ↪ (Spanish notation)
FUNCTION normalise_date(v)    # "12-05-2026", "12 de mayo de 2026" → a
    ↪ date
FUNCTION normalise_text(v)    # lower case, no accents, single spaces

# ================================================================
# ONE CHECK AS AN EXAMPLE, to show the shape shared by all of them
# ================================================================
# AGE-01: the age of the building in the report must match the year of
# construction in the cadastral record.
FUNCTION check_age(report, request, settings) RETURNS CheckResult:
    (cadastre, ev_cad) ← read_field(report, "cadastral_record.age")
    IF cadastre IS NULL:
```

```
        RETURN NO_DATA("the cadastral record gives no year of
    ↪ construction")

    (stated, ev_rep) ← read_field(report, "building_description.age")
    IF stated IS NULL:
        (stated, ev_rep) ← read_field(report, "cost_method.age")   #
    ↪ the other place it appears
    IF stated IS NULL:
        RETURN NO_DATA("the report gives no age for the building")

    # The report may give an age in years and the cadastre a year: both
    ↪ are
    # converted to a year of construction before being compared.
    IF difference(stated, cadastre) > settings.AGE_TOLERANCE_YEARS:
        RETURN KO("age does not match: report {stated} vs cadastre
    ↪ {cadastre}",
                  evidences=[ev_rep, ev_cad])
    RETURN OK("age matches ({cadastre})", evidences=[ev_rep, ev_cad])

# Note what the result carries: not just OK or KO, but the two values
    ↪ compared
# and the literal quotations they came from. The reviewer can check the
# conclusion without opening the PDF.
```

Pseudocode F.4: Block 04, validations and result

F.5 Orchestrator: Running the Blocks as a Chain

```
# ================================================================
# ORCHESTRATOR
# Runs the four blocks as a chain for ONE document: the output of a
# block is the input of the next one. It holds NO business logic: it
# does not know what an appraisal report is, and it decides nothing
# about the document. It only knows the order of the blocks and how
# to read the state each one returns.
# ================================================================

```

```
# The chain in order. Each block declares which of its states allow the
# document to continue (see the state contract).
CONSTANT CHAIN = [ Block01, Block02, Block03, Block04 ]

METHOD run(file_name) RETURNS Trace:
    trace ← NEW Trace()          # what happened at every step, for the
    ↪ record

    # 1) The first block is the only one that receives a file name: it
    ↪ is the
    #    entry point of the pipeline and the one that computes the
    ↪ identifier.
    result ← Block01.run(file_name)
    trace.add(block="Block 01", state=result.state)
    IF result.state NOT IN Block01.CONTINUES:
        RETURN trace             # this document stops here; the run
    ↪ does not break
    sha256 ← result.sha256       # from now on, the document travels as
    ↪ its identifier

    # 2) The rest of the chain works on that identifier.
    FOR EACH block IN [ Block02, Block03, Block04 ]:
        result ← block.run(sha256)
        trace.add(block=block.name, state=result.state)
        IF result.state NOT IN block.CONTINUES:
            # Either an expected stop (a scanned PDF, an appraiser with
            ↪ no prompt)
            # or an error. In both cases the next block would have no
            ↪ input, so
            # there is nothing to gain by calling it. The state says why.
            RETURN trace

    RETURN trace                 # the document reached the end of the
    ↪ pipeline

# ---- Notes ------------------------------------------------------
```

```
# * A stop ends the path of THIS document, not the run: another document can be
#   sent through the chain straight away.
# * Re-running a document that stopped is safe and cheap: each block reports its
#   previous work as an "already done" state instead of repeating it, so only the
#   step that stopped is actually retried.
# * On the cloud this method is replaced by a managed orchestration (Azure Durable
#   Functions or Logic Apps) triggered by the arrival of a document. Because the
#   decision is taken from a state, and not from a process exit code, the
#   substitution does not require any change inside the blocks.
```

Pseudocode F.5: Orchestrator, running the four blocks as a chain

Acknowledgments

I would like to thank Instituto de Análisis Inmobiliario (Instai), part of the Euroval group, for the opportunity to develop the REVIN project during my internship, and for trusting me with its design and its implementation.

I am especially grateful to my company tutor, Ignacio Ribelles Ferrándiz, for his guidance and for the way he taught: he did not give me ready-made answers, but encouraged me to investigate and to learn from my own mistakes. I also thank the rest of the data team for their help and for making the office a good place to work.

I thank my academic supervisor at the University of Alicante, Pedro Albarrán Pérez, of the Department of Economics (Fundamentos del Análisis Económico), for the support during the preparation of this thesis.

I thank my parents, who gave me the support that made it possible for me to take on this internship in the first place.

And I thank my girlfriend, for her patience, her support and her understanding when I decided to take on this project at a moment that mattered a great deal in her own life.

Finally, I thank the rest of my family and my friends for their support throughout the master's degree.