Facultad de Ingeniería
Ingeniaritza Fakultatea

Grado en Ingeniería Informática

Informatikako Ingeniaritzako gradua

Proyecto fin de grado

Gradu amaierako proiektua

Design and implementation of a reproducible web-based platform for experimentation and comparative evaluation of time series forecasting models with semantic context integration via large language models and the Model Context Protocol

Diego Ramírez Lacalle

Directora: Nicole Cusimano

Bilbao, julio de 2026

Grado en Ingeniería Informática

Informatikako Ingeniaritzako gradua

Proyecto fin de grado

Gradu amaierako proiektua

Design and implementation of a reproducible web-based platform for experimentation and comparative evaluation of time series forecasting models with semantic context integration via large language models and the Model Context Protocol

Diego Ramírez Lacalle

Directora: Nicole Cusimano

Bilbao, julio de 2026

Abstract

This Final Degree Project presents the software-engineering component of an integrated project on inflation forecasting with contextual economic signals. The work focuses on the design and implementation of the system that supports the forecasting study. Its main goal is to move the project beyond isolated scripts and notebooks by providing a structured environment where the platform entities and user workflows can be managed in a reproducible way.

The architecture connects the forecasting code with backend services, a web frontend, persistent storage, a gateway, experiment tracking, and a Model Context Protocol server. The backend exposes the main platform workflows through a REST API. The frontend supports experiment creation, run inspection, model comparison, forecast visualization, and contextual exploration. Docker-based orchestration runs the multi-service stack as a complete execution environment.

The implemented outcome includes 84 backend integration tests passing in the Compose environment, a drift endpoint verified with real data through a Kolmogorov–Smirnov residual test, and a multi-service stack orchestrated with Docker Compose.

The contribution of this thesis is the engineering layer that makes the forecasting work operational: repository organization, separation of responsibilities between services, integration of forecasting adapters and MCP-derived context, persistence of experiments and results, and validation mechanisms for a maintainable platform. The thesis explains how models and signals can be executed, inspected, compared, and evolved within a coherent software architecture.

Descriptors

Software architecture, Inflation forecasting, MCP, Web platform, Reproducibility

Contents

1. Introduction **1**

2. Background and Justification **4**

2.1. Economic context 4
2.2. Current state-of-the-art 4
2.2.1. From forecasting experiments to operational ML systems 5
2.2.2. Technical debt and maintainability 5
2.2.3. API-based architecture 6
2.2.4. Experiment tracking and reproducibility 6
2.2.5. Containerized deployment 7
2.2.6. Contextual integration through MCP 7
2.2.7. Monitoring and drift awareness 8
2.2.8. Engineering gap 8
2.3. Technical justification 8
2.4. Economic and social justification 9
2.5. Professional relevance in the banking sector 10
2.6. Project opportunity 10

3. Objectives and Scope **12**

3.1. General objective 12
3.2. Specific objectives 12
3.3. Engineering questions 13
3.4. Functional scope 13
3.5. Scope of the INF contribution 14
3.6. Out of scope 15
3.7. Expected contribution 16

4. Planning 17
4.1. Planning approach 17
4.2. Task definition 17
4.2.1. General tasks 18
4.2.2. CDIA-specific tasks 19
4.2.3. INF-specific tasks 19
4.3. Milestones 19
4.4. Task distribution 19
4.5. Human-resource plan 20
4.6. Planning risks 23

5. Budget 24
5.1. Human resources 24
5.2. Materials and equipment 25
5.3. Total budget 26
5.4. Interpretation 26

6. Methodology 27
6.1. Methodological approach 27
6.2. Connection with the forecasting workflow 28
6.3. Requirements analysis 28
6.4. Architecture design method 29
6.5. Iterative implementation 30
6.6. Integration strategy 31
6.7. Validation strategy 31
6.8. Documentation and reproducibility 32

7. Development 33
7.1. Purpose of the chapter 33
7.2. Development scope 33
7.3. Requirements implemented 34
7.3.1. Functional requirements 34
7.3.2. Non-functional requirements 36
7.4. Repository organization 37
7.5. General architecture 38

7.6. Backend and API 40
7.6.1. Application entry point 40
7.6.2. Router organization 40
7.6.3. Experiment lifecycle 42
7.6.4. Authentication and permissions 43
7.6.5. Assistant and narrative analysis 44
7.7. Data model and persistence 46
7.7.1. Relational entities 46
7.7.2. Dataset and model catalogue 48
7.7.3. Experiment and run outputs 48
7.7.4. MongoDB context storage 48
7.8. Forecasting adapter layer 49
7.8.1. Adapter contract 49
7.8.2. Adapter registry 51
7.8.3. Execution flow 52
7.9. MCP integration 53
7.9.1. MCP server tools 53
7.9.2. Backend MCP client 55
7.9.3. MCP-derived exogenous features 55
7.9.4. Context storage and traceability 56
7.10. Frontend application 56
7.10.1. Application routing 56
7.10.2. API access and state management 58
7.10.3. Experiment creation workflow 58
7.10.4. Run inspection 59
7.10.5. Model comparison dashboard 59
7.10.6. Simulation and context views 60
7.10.7. Frontend design decisions 60
7.11. Execution environment and reproducibility 60
7.11.1. Compose services 61
7.11.2. Configuration management 62
7.11.3. Containers and mounted code 62
7.11.4. Gateway and network boundary 63
7.11.5. Persistent volumes 63

7.11.6. Experiment tracking with MLflow 63
7.11.7. Reproducibility limits 65
7.12. Validation and testing 65
7.12.1. Test organization 65
7.12.2. Unit-level checks 67
7.12.3. Database and fixture isolation 67
7.12.4. Run lifecycle validation 67
7.12.5. Metric and comparison validation 68
7.12.6. Operational checks and drift 68
7.12.7. Continuous integration pipeline 69
7.12.8. Validation limits 69
7.13. Connection with the forecasting pipeline 70
7.13.1. Integration principle 71
7.13.2. Processed data artifacts 71
7.13.3. Dataset and catalogue seeding 72
7.13.4. Experiment conditions and MCP connection 73
7.13.5. Result artifacts and platform outputs 74
7.13.6. Shared utilities 74
7.13.7. Traceability across platform layers 74
7.14. Maintainability and extension points 75
7.14.1. Module boundaries 75
7.14.2. Security and access-control considerations 76
7.14.3. Operational robustness 76
7.14.4. Extension points 77
7.14.5. Development limits 78
7.15. User workflows and operating guide 78
7.15.1. Initial execution workflow 78
7.15.2. Authentication and navigation 79
7.15.3. Experiment workflow 79
7.15.4. Run inspection workflow 80
7.15.5. Comparison workflow 80
7.15.6. Simulation and context workflows 81
7.15.7. Workflow summary 81

7.16. Interface evidence . . . 82
7.16.1. Experiment management screens . . . 82
7.16.2. Run inspection screens . . . 83
7.16.3. Comparison dashboard screens . . . 83
7.16.4. Simulation and context screens . . . 84
7.16.5. Interface visual evidence status . . . 85
7.17. Requirements traceability . . . 85
7.17.1. Functional coverage . . . 86
7.17.2. Non-functional coverage . . . 86
7.17.3. Coverage limits . . . 87

8. Ethical Assessment **88**
8.1. Purpose of the assessment . . . 88
8.2. Responsible use of economic forecasts . . . 88
8.3. Transparency and traceability . . . 89
8.4. Human oversight and automation limits . . . 89
8.5. Data governance and privacy . . . 90
8.6. Fairness, bias, and representativeness . . . 90
8.7. Security and misuse risks . . . 91
8.8. Environmental and resource considerations . . . 91
8.9. Mitigation summary . . . 92
8.10. Residual responsibility . . . 93

9. Incidents **94**
9.1. Purpose of the chapter . . . 94
9.2. TimeGPT API dependency . . . 94
9.3. Optional model packages . . . 94
9.4. MCP and news-service availability . . . 95
9.5. Test isolation and database state . . . 95
9.6. State-of-the-art material for the INF thesis . . . 96
9.7. Scope adjustment of production features . . . 96
9.8. Incident summary . . . 97

10. Conclusions and Future Work **98**
10.1. Final assessment . . . 98

10.2. Fulfilment of objectives 98
10.2.1. Engineering-question verdicts 99
10.3. Engineering contribution 100
10.4. Limitations 100
10.5. Future work 101
10.5.1. Extend validation evidence 101
10.5.2. Production hardening 102
10.5.3. Continuous integration and automated quality 102
10.5.4. Monitoring and model governance 102
10.5.5. Improved MCP and contextual integration 102
10.5.6. More complete user experience 102
10.5.7. Extension of models and datasets 103
10.6. Closing statement 103

11. Bibliography **104**

Definitions, Acronyms, and Abbreviations **108**
Acronyms and abbreviations 108
Key definitions 112

A. Appendices **114**
A.1. Repository and service map 114
A.2. Execution reminder 115
A.3. API endpoint families 116
A.4. Validation reminder 117
A.5. Declaration of Artificial Intelligence Use 118
A.6. Final evidence checklist 119

List of Figures

3.1. Functional scope of the INF platform and its relation to the forecasting workflow. 14

4.1. Integrated Gantt diagram distinguishing MVPs, user stories, degree profiles, and estimated effort 21

6.1. Methodological workflow followed in the INF contribution. 27

7.1. General architecture of the INF platform. 38
7.2. Run execution flow and lifecycle states. 43
7.3. Main persistence model of the INF platform. 46
7.4. Model execution layer exposed by the INF platform. 52
7.5. MCP signal retrieval and storage flow in the INF platform. 54
7.6. Frontend navigation and main user workflows. 57
7.7. Integration boundary between the forecasting pipeline and the INF platform. 70
7.8. Main user workflow from platform start-up to result inspection. 79
7.9. Experiment list and new-experiment creation workflow. 82
7.10. Run-detail view with experiment metadata, completed run status, and drift warning. 83
7.11. Model comparison dashboard for same-series forecasting experiments. 83
7.12. What-if simulator with signal levers, horizon selection, forecast chart, and local assistant panel. 84
7.13. Inflation-pulse page with FinBERT sentiment summary and live inflation-related headlines. . . 84

List of Tables

4.1. User stories and main tasks of the integrated project . . . 18
4.2. Estimated workload distribution by phase . . . 20
4.3. Roles, responsibilities, and assigned personnel . . . 22
4.4. Estimated workload distribution by role . . . 22

5.1. Estimated salary expenses . . . 24
5.2. Human-resource cost by project profile . . . 25
5.3. Estimated materials and equipment costs . . . 25
5.4. Budget summary . . . 26

7.1. Functional requirements of the INF platform . . . 35
7.2. Non-functional requirements of the INF platform . . . 36
7.3. Repository areas used by the INF platform . . . 37
7.4. Main components of the developed platform . . . 39
7.5. Backend API route groups . . . 41
7.6. Main PostgreSQL entities . . . 47
7.7. Forecast adapter contract . . . 50
7.8. Forecasting adapters implemented in the backend . . . 51
7.9. MCP tools exposed by the platform . . . 53
7.10. Main frontend routes . . . 57
7.11. Services defined in the Docker Compose environment . . . 61
7.12. Main backend validation areas . . . 66
7.13. Main forecasting artifacts consumed by the INF platform . . . 72
7.14. Main extension points of the INF platform . . . 77
7.15. Main user-facing workflows of the INF platform . . . 81
7.16. Interface visual evidence for the Development chapter . . . 85
7.17. Traceability of functional requirements . . . 86

7.18. Traceability of non-functional requirements 87

8.1. Ethical risks and mitigation mechanisms 92

9.1. Summary of incidents and responses 97

A.1. Repository areas relevant to the INF thesis 114
A.2. Service stack used by the platform 115
A.3. Relevant configuration variables 116
A.4. Main backend endpoint families 116
A.5. Backend validation areas 117
A.6. Use of artificial intelligence tools during the INF project 118
A.7. Final interface evidence status 120

List of Pseudocodes

7.1. Example experiment-creation request and response shape 42
7.2. Inputs and output of the run narration endpoint 45
7.3. MongoDB document shape for stored MCP context 49
7.4. Example macro-signal record returned for 2024-12 54
7.5. MLflow parameters and metrics logged for one run 64

A.1. Basic local execution reminder 115
A.2. Backend test execution reminder 117

1. INTRODUCTION

Economic forecasting is the process of using historical data, current indicators, and explicit assumptions to estimate the future behaviour of economic variables. These variables may include inflation, employment, interest rates, gross domestic product, exchange rates, consumption, prices, or financial conditions. Its main objective is to reduce uncertainty enough to support better decisions. In practice, economic forecasts are used to anticipate risks, compare scenarios, plan resources, evaluate policy alternatives, and detect changes in economic trends before they are fully visible in official statistics.

Inflation forecasting is one of the most relevant branches of economic forecasting. Inflation affects households, companies, public institutions, and financial markets because it changes the real value of income, savings, costs, and future decisions. Forecasting inflation is therefore both a technical and practical problem. A useful forecasting system can help anticipate possible changes, compare scenarios, and support better economic analysis. At the same time, inflation is difficult to model: it depends on local consumption patterns, monetary policy, energy prices, supply chains, expectations, and external shocks. The period after 2020 made this especially visible, with the combined effect of the pandemic, supply disruptions, and the energy shock in Europe.

This Final Degree Project has been developed as an integrated project with two complementary theses. The Computer Engineering (INF) part focuses on the software and system-engineering side: architecture, integration, deployment, and the organization of the platform. The Data Science and Artificial Intelligence (CDIA) part focuses on the data science and artificial intelligence side. Its purpose is to study whether modern forecasting models, especially time-series foundation models, can improve inflation forecasts when compared with classical statistical approaches, and whether external contextual signals add useful information to the prediction process.

In this thesis, time-series foundation models are understood as forecasting models pretrained on large and diverse collections of time series and later applied to a new forecasting problem with little or no task-specific training. This idea follows the recent research line on foundation models for time-series analysis, where the model is expected to transfer patterns learned from many temporal datasets to new forecasting tasks [1]. Contextual signals, on the other hand, refer to external variables that are not the target inflation series itself, but may help explain its evolution, such as macroeconomic indicators, energy prices, uncertainty measures, institutional communication, or text-derived information.

The central research question is therefore:

> Do foundation time-series models and contextual signals improve inflation forecasting, and under which conditions does this improvement appear?

The project studies three monthly inflation series with different scopes: the Spanish Consumer Price Index (Spain CPI), a global Consumer Price Index indicator (Global CPI), and the European Harmonised Index of Consumer Prices (European HICP). This selection is important because the usefulness of a forecasting model or an external signal may depend on the scale of the target variable. A signal that is relevant for broad global inflation may explain domestic Spanish inflation with less strength. In the same way, European institutional communication may be more directly related to European HICP than to a global aggregate. The project therefore compares several inflation contexts in the same experimental framework.

The models evaluated in the project are grouped into three families. First, classical statistical models such as Autoregressive Integrated Moving Average (ARIMA), Seasonal Autoregressive Integrated Moving Average (SARIMA), Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors (SARIMAX), and automatic ARIMA (AutoARIMA) are used as strong baselines. They are included not as simple reference models, but as serious competitors, because in many economic time series they remain difficult to beat. Second, deep learning models such as Long Short-Term Memory (LSTM), Neural Basis Expansion Analysis for Time Series (N-BEATS), and Neural Hierarchical Interpolation for Time Series (N-HiTS) are considered to test whether neural models trained on the available data improve the forecasts. Third, the project evaluates foundation time-series models such as Chronos-2, TimesFM, and TimeGPT. These models are relevant because they are pretrained on large collections of time series and are designed to generalize to new forecasting tasks with limited task-specific training.

In addition to comparing model families, the project studies the role of contextual information. The baseline condition, named `C0`, uses only the historical target series. The contextual conditions, grouped under `C1`, add exogenous information, including institutional variables, macroeconomic indicators, energy prices, uncertainty measures, geopolitical risk, and text-derived signals processed through the Model Context Protocol (MCP) pipeline. These conditions are separated into `C1_inst` (institutional and macroeconomic context), `C1_mcp` (MCP and text-derived context), and `C1_full` (the combined context), so the analysis can distinguish between institutional and macroeconomic context, text-based signals, and their combination. This distinction is important because context is useful when it is available at the right time, aligned with the target, and relevant to the dynamics being predicted.

A rolling-origin evaluation is an out-of-sample forecasting procedure in which the forecast origin moves forward through time and the model is evaluated repeatedly using only the information available before each prediction. This type of design is commonly used in forecast evaluation because it is closer to the way a forecasting

system would operate in practice than a random train-test split [2].

The evaluation follows a rolling-origin backtesting design over the 2021–2024 period, with forecasts at 1, 3, 6, and 12-month horizons. This design is used to simulate a realistic forecasting situation: at each origin, the model can only use information that would have been available at that date. The main metrics used in the project are Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Mean Absolute Scaled Error (MASE). MASE is especially useful because it compares the model against a seasonal naive benchmark and makes the interpretation less dependent on the scale of each series [3]. Where appropriate, Diebold-Mariano tests are used to compare predictive performance more formally [4].

The expected contribution of the CDIA part is to identify when foundation models help, when classical forecasting remains stronger, and how this depends on the series, the forecast horizon, and the type of external context. The current evidence of the project points to a nuanced result: classical models remain highly competitive, especially at short horizons and for Spain CPI, while foundation models combined with relevant context show more value for global and European inflation at longer horizons. This makes the project useful as a model comparison and as an empirical study of the limits of contextual and foundation-model forecasting.

The expected contribution of the INF part is to turn that forecasting work into a software platform. The platform gives operational form to the project through a multi-service stack, forecasting adapters, drift analysis, and user-facing workflows for creating experiments, inspecting runs, comparing metrics, and exploring contextual information.

The rest of the thesis is organized as follows. Chapter 2 presents the background and justification of the project from a software-engineering perspective, including MLOps, API-based systems, reproducible deployment, MCP integration, and monitoring. Chapter 3 defines the INF objectives, engineering questions, functional scope, and limits of the platform. Chapters 4 and 5 describe the planning and budget. Chapter 6 explains the methodology followed to translate the forecasting workflow into software requirements, architecture, integration, validation, and documentation. Chapter 7 presents the development of the platform: requirements, backend, data model, forecasting adapters, MCP integration, frontend, persistence, deployment, validation, traceability, and user workflows. Chapter 8 discusses the ethical implications of exposing AI-based forecasts through a software system. Chapter 9 records the main incidents and scope adjustments found during development. Chapter 10 summarizes the engineering verdict and proposes future lines of work. The final parts collect the definitions, acronyms, bibliography, and appendices needed to support the main text.

2. BACKGROUND AND JUSTIFICATION

2.1. ECONOMIC CONTEXT

Inflation is one of the macroeconomic variables with the strongest effect on everyday economic decisions. It changes purchasing power, affects saving and consumption decisions, modifies production costs, and influences interest rates, wages, investment, and public policy. For households, inflation determines how much real income is preserved over time. For companies, it affects pricing, margins, inventory decisions, and financial planning. For banks and other financial institutions, inflation is connected to interest-rate expectations, credit risk, market conditions, and the interpretation of macroeconomic scenarios. For public institutions, inflation forecasts are a key input in policy design because many decisions must be taken before the full effect of current economic conditions is visible.

This makes inflation forecasting both useful and difficult. It is useful because anticipating inflation helps decision makers prepare for possible future changes. It is difficult because inflation is not driven by a single mechanism. It can respond to domestic demand, energy prices, supply-chain disruptions, monetary policy, exchange rates, expectations, geopolitical uncertainty, and communication from economic institutions. In addition, the relevance of these factors changes over time. The post-2020 period is a clear example: the pandemic, supply bottlenecks, energy-market tensions, and the European inflation shock created conditions where historical patterns alone were often not enough to explain short and medium-term movements.

For this reason, a forecasting project in this area should ask which model obtains the lowest error and under which conditions that model is reliable. A model that works for a global inflation aggregate may fail for a national CPI series. A signal that is useful during an energy shock may be irrelevant in a stable period. A complex model may improve long-horizon forecasts but add little value at one month. This project is justified by the need to study these differences in a structured and reproducible way.

2.2. CURRENT STATE-OF-THE-ART

The state-of-the-art relevant to the INF thesis is not centred on forecasting accuracy itself, because that question belongs mainly to the CDIA contribution. From the Computer Engineering perspective, the relevant question is how a forecasting study that combines datasets, model families, contextual signals, metrics, and user-facing results can be transformed into a software system that is executable, traceable, maintainable, and extensible.

This places the INF contribution in the area where machine-learning experimentation, software architecture, data engineering, deployment, and monitoring meet.

2.2.1. From forecasting experiments to operational ML systems

Many applied machine-learning projects begin with exploratory work: notebooks, scripts, local datasets, manual result files, and iterative model testing. This is a natural way to investigate a problem, especially when the first objective is to understand the data and validate the modelling idea. The problem appears when the experimental work starts to involve several datasets, multiple model families, repeated executions, external services, result comparisons, and user-facing outputs. At that point, the project needs a system around the models.

The MLOps literature describes this transition. Kreuzberger, Kuehl, and Hirschl define MLOps as a paradigm that combines machine learning, software engineering, and data engineering in order to operationalize ML systems [5]. This idea is directly relevant to this integrated project. The INF platform does not try to reproduce a large industrial MLOps environment, but it follows the same basic need: forecasting experiments should be executable, traceable, comparable, and maintainable.

Google's MLOps guidance also emphasizes that machine-learning systems involve more than a trained model. Operational workflows include data preparation, validation, deployment, monitoring, and feedback loops [6]. In this project, these concerns appear in a smaller but concrete form. The backend controls experiments and runs. The adapter layer exposes model execution. MLflow records experiment information. Docker Compose coordinates the execution environment. The frontend exposes the results to users. The drift endpoint adds a first monitoring mechanism. Together, these elements turn the forecasting work into a platform instead of a collection of disconnected artifacts.

2.2.2. Technical debt and maintainability

The need for a platform is also justified by the technical debt that appears in machine-learning systems. Sculley et al. argue that ML systems combine the maintenance problems of traditional software with additional risks caused by data dependencies, configuration, entanglement, hidden feedback loops, and weak system boundaries [7]. This is particularly relevant in forecasting projects, where the final prediction depends not only on the model code but also on data alignment, temporal splits, exogenous variables, scaling decisions, and stored outputs.

In the IPC-MCP project, this risk is visible in several places. Forecasting scripts must use the correct histori-

cal window. Contextual signals must be aligned with the forecast origin and must not leak future information. Different model families produce outputs that need to be compared with common metrics. Results must remain linked to the experiment that generated them. The platform addresses these risks by introducing explicit boundaries: forecasting adapters isolate model execution, backend services manage workflows, databases store structured entities and context, the MCP server exposes external information, and the frontend presents the results without depending on notebook internals.

This does not remove every source of technical debt, but it changes the nature of the project. Future work can add models, views, signals, or validation mechanisms through defined parts of the architecture instead of modifying an unstructured set of scripts. That is one of the main engineering reasons for developing the INF part of the integrated project.

2.2.3. API-based architecture

The backend of the platform follows an API-based organization. REST, as introduced by Fielding, is an architectural style for network-based systems that emphasizes constraints such as client-server separation, stateless communication, uniform interfaces, and layered organization [8]. These ideas fit the project because the frontend should not need to know the internal details of each forecasting script or model adapter. It should interact with experiments, runs, datasets, metrics, and contextual information through stable service endpoints.

OpenAPI complements this approach by providing a standard way to describe HTTP APIs so that both humans and software tools can understand the available operations [9]. This is useful in the project because the backend is not only an internal implementation detail. It is the contract between the platform's user interface and the forecasting system. A clear API makes the architecture easier to document, test, and extend.

2.2.4. Experiment tracking and reproducibility

Reproducibility in the CDIA thesis is mainly methodological: temporal splits, rolling-origin evaluation, metrics, leakage prevention, and fair comparison. In the INF thesis, reproducibility also has a system meaning. The platform should make it possible to know which experiment was created, which model was used, which horizon was selected, whether MCP context was enabled, what predictions were produced, and which metrics were obtained.

MLflow is relevant here because it was designed to support the machine-learning lifecycle, especially experiment tracking, reproducibility, and deployment flexibility [10]. In this project, MLflow provides a first tracking

layer for model runs and metrics. It is not the only persistence mechanism, because the platform also stores structured application entities in PostgreSQL, but it reinforces the idea that experiments should leave a trace beyond the immediate execution.

2.2.5. Containerized deployment

The platform is composed of several services: backend, frontend, gateway, PostgreSQL, MongoDB, MLflow, and MCP server. Running these components manually would make the system more fragile and less reproducible. Docker addresses part of this problem by packaging applications and dependencies into lightweight containers, which helps reduce environment inconsistencies between machines [11]. Docker Compose then provides a practical way to define and run multi-container applications from a single configuration file [12].

For the INF thesis, deployment is part of the architecture rather than an afterthought. PostgreSQL stores structured platform data, MongoDB stores news and contextual documents, MLflow tracks experiments, the MCP server exposes contextual tools, the backend coordinates workflows, and the frontend provides user interaction. Docker Compose connects these parts into a complete environment that can be understood and reproduced more easily.

2.2.6. Contextual integration through MCP

The CDIA thesis studies whether contextual information improves forecasting. The INF thesis focuses on how that contextual information is integrated into the system. The Model Context Protocol (MCP) is relevant because it defines a client-server pattern for connecting AI applications to external tools and data sources [13]. In this project, the MCP server exposes macroeconomic signals, news context, and sentiment information, while the backend acts as the client that retrieves this context for the relevant forecast timestamps.

This design matters because contextual signals are expected to evolve. Future versions of the project may add new sources, new sentiment methods, new countries, or new macroeconomic indicators. By treating context retrieval as a service interface, the platform avoids embedding every external-data decision directly into the frontend or into each forecasting adapter. It also makes the architecture more consistent with the central motivation of the project: connecting the forecasting study with a system that can grow around it.

2.2.7. Monitoring and drift awareness

A forecasting platform should not only generate predictions. It should also support inspection of model behaviour over time. Drift is one of the main operational risks in machine-learning systems because the data observed during use may differ from the data used during training or previous evaluation. Drift can affect the input distribution, the relationship between inputs and targets, or both [14]. Large-scale ML systems can experience performance degradation even when retraining or model selection strategies are in place [15].

The platform includes a first drift-analysis endpoint based on comparing residual distributions in completed runs. This is not a complete industrial monitoring solution, and it should not be presented as one. Its value is more precise: it introduces the engineering habit of checking whether forecast behaviour changes over time. In an economic forecasting project, this is especially relevant because inflation dynamics can change during shocks, policy transitions, or energy-market disruptions.

2.2.8. Engineering gap

The gap addressed by the INF thesis is the conversion of a rigorous forecasting experiment into a structured software platform. The CDIA part studies the empirical question: whether foundation time-series models and contextual signals improve inflation forecasting. The INF part studies the engineering question: how to organize the platform entities and workflows coherently.

This gap is relevant because good forecasting evidence is not automatically a usable system. Without explicit software boundaries, experiments become fragile, results are hard to trace, and contextual integrations are difficult to extend. The INF contribution therefore complements the CDIA contribution by giving the forecasting work an operational form through APIs, persistence, adapters, containerized services, user workflows, and validation mechanisms.

2.3. TECHNICAL JUSTIFICATION

From a technical point of view, the project is justified by three needs. The first is the need for a fair comparison between model families. If foundation models are compared only with weak baselines, their value can be overstated. For this reason, the project includes ARIMA, SARIMA, SARIMAX, and AutoARIMA models, together with deep learning alternatives such as LSTM, N-BEATS, and N-HiTS. This makes it possible to evaluate whether improvements come from the foundation-model approach itself, from the use of contextual variables, or simply

from the structure of the target series.

The second need is to separate the effect of different types of context. The project distinguishes between `C0`, where the model only uses the target series history, and several `C1` configurations where external information is added. Institutional and macroeconomic signals are separated from MCP or text-derived signals, and in some cases they are also combined. This design is important because a global conclusion such as "context helps" would be too vague. Context may help for one series and harm another. It may help at 12 months but not at 1 month. It may improve TimesFM but not TimeGPT. The project therefore evaluates context as an empirical question whose answer depends on the target, horizon, and model family.

The third need is methodological reliability. The evaluation uses rolling-origin backtesting over the 2021–2024 period and forecast horizons of 1, 3, 6, and 12 months. This is closer to a real forecasting situation than a single train/test split, because each origin simulates the information available at a given point in time. The project also applies safeguards such as shifting exogenous signals so future information is not leaked into the model, scaling variables before residual correction, and using MASE to compare performance against a seasonal naive benchmark. These choices make the experiment more credible and reduce the risk of obtaining optimistic results from an unrealistic setup.

2.4. ECONOMIC AND SOCIAL JUSTIFICATION

The economic justification of the project comes from the value of better inflation analysis. Inflation forecasts are relevant for planning, budgeting, pricing, financial decisions, and policy discussion. Even a modest improvement can be useful if it occurs in the right horizon and for the right target. Long-horizon forecasts are especially relevant because many decisions in finance, business, and policy are made months before their consequences are observed. The project therefore studies when model complexity is justified and when simpler approaches remain more reliable.

The social justification is related to transparency and responsible interpretation. Inflation affects the cost of living and can become a source of uncertainty for society. Forecasting systems should therefore be presented with their limitations. A model may be useful without being definitive. A signal may be informative without being stable across all periods. A foundation model may perform well in one context and fail in another. By comparing several model families, targets, horizons, and signal types, this project supports a more careful view of AI-based economic forecasting.

2.5. PROFESSIONAL RELEVANCE IN THE BANKING SECTOR

This topic also has direct professional relevance for me because I am currently working in a bank, where inflation, interest rates, macroeconomic expectations, and risk conditions are part of the broader environment in which many decisions are interpreted. In a banking context, forecasting is a practical tool for scenario analysis, risk monitoring, market interpretation, customer and portfolio analysis, and the evaluation of macroeconomic assumptions used by different business areas.

This relevance became especially clear in a professional conversation with Pedro Gómez Tejerina, my professional supervisor. His profile is especially close to the topic of this project: he is a Senior Data Science Expert at BBVA, has more than 22 years of experience in the bank, and also teaches Business Intelligence and Big Data at the University of Deusto [16]. From that position, he highlighted the importance of inflation forecasting for banking activity, data-driven analysis, and market-oriented decision making:

> “Current forecasting models are not always fully reliable for market decisions. In a bank that works with financial markets, anticipating inflation, interest rates, and macroeconomic movements is essential, and we need to remain aware of how technology is advancing in this area.”

This comment complements the academic evaluation of the project and helps explain its applied relevance. Financial institutions need to anticipate changes in the economic environment and understand the limits of the models they use. A forecast based on a more advanced model still requires evidence of reliability. For that reason, this project is relevant to the banking sector because it compares modern AI forecasting models with classical baselines, studies whether contextual signals add value, and identifies the situations in which the resulting forecasts should be treated with caution.

2.6. PROJECT OPPORTUNITY

The opportunity of this project is therefore both practical and methodological. Practically, it addresses a real forecasting problem with clear economic relevance. Methodologically, it combines model comparison, contextual-signal engineering, and reproducible evaluation. I do not approach the problem as a search for the most modern model by default. The project asks whether different sources of information actually improve prediction, and it measures that improvement across series and horizons.

This distinction is important for the final interpretation of the thesis. If the results show that context helps in Global CPI and Europe HICP but not in Spain CPI, that is not a failure of the project. It is one of its main findings.

The value of the work lies in identifying the conditions under which foundation models and semantic context are useful, and also the conditions under which classical models remain stronger. This makes the project a realistic contribution to applied economic forecasting: it tests a modern idea, but it does so with baselines, controls, and limitations that make the conclusion more credible.

3. OBJECTIVES AND SCOPE

3.1. GENERAL OBJECTIVE

The general objective of the INF part is to design and implement a modular INF platform for executing, storing, inspecting, and extending the inflation-forecasting workflow.

This objective is centred on software engineering. The system must organize the main platform entities, contextual integration, persistence, visualization, and deployment through backend services and user-facing workflows.

3.2. SPECIFIC OBJECTIVES

The general objective is divided into the following specific objectives:

1. Analyse the needs of the forecasting project from a software-engineering perspective, identifying which parts of the workflow must be exposed, stored, executed, compared, or visualized by the platform.
2. Design a modular architecture for the INF platform, separating backend services, frontend interaction, gateway configuration, persistence, forecasting execution, MCP integration, and deployment infrastructure.
3. Implement a backend API capable of managing the main application entities and platform workflows.
4. Integrate the forecasting logic through adapters for the model families used by the platform.
5. Connect the platform with contextual economic information through the Model Context Protocol (MCP), allowing macroeconomic signals, news context, and sentiment information to be retrieved as part of the forecasting workflow.
6. Define a persistence layer that separates structured platform entities from semi-structured contextual information, using PostgreSQL and MongoDB according to the data shape.
7. Build a frontend interface that allows users to access the platform, create and inspect experiments, review forecast runs, compare models, visualize predictions, explore contextual information, and use simulation

views.

8. Provide a reproducible execution environment through Docker Compose, coordinating the backend, frontend, gateway, PostgreSQL, MongoDB, MLflow, and MCP server as a complete service stack.

9. Include validation and observability mechanisms appropriate to the scope of the project, such as health checks, integration tests, metric storage, MLflow logging, and a first drift-analysis endpoint.

10. Document the scope, limits, and future extension points of the platform.

3.3. ENGINEERING QUESTIONS

In order to reach the proposed objectives, the INF work is guided by the following engineering questions:

1. How can the forecasting workflow be connected to a software platform without losing traceability between datasets, models, runs, predictions, and metrics?

2. How should the system separate responsibilities between backend, frontend, persistence, forecasting adapters, contextual signal retrieval, and deployment?

3. How can contextual information produced or exposed through MCP be integrated as a software capability without embedding unsupported model-performance claims in the platform?

4. How can the platform support forecast-result inspection, comparison, and interpretation through user-facing workflows?

5. How can the architecture remain extensible enough to add new models, datasets, signals, views, and validation mechanisms in future work?

3.4. FUNCTIONAL SCOPE

The functional scope of the INF contribution covers the main platform capabilities. It includes user access, dataset and model catalogue management, experiment creation, forecast run execution, metric storage, result inspection, model comparison, contextual news and signal retrieval, drift analysis, and simulation workflows.

The backend is responsible for exposing the platform workflows through a REST API. This includes the applica-

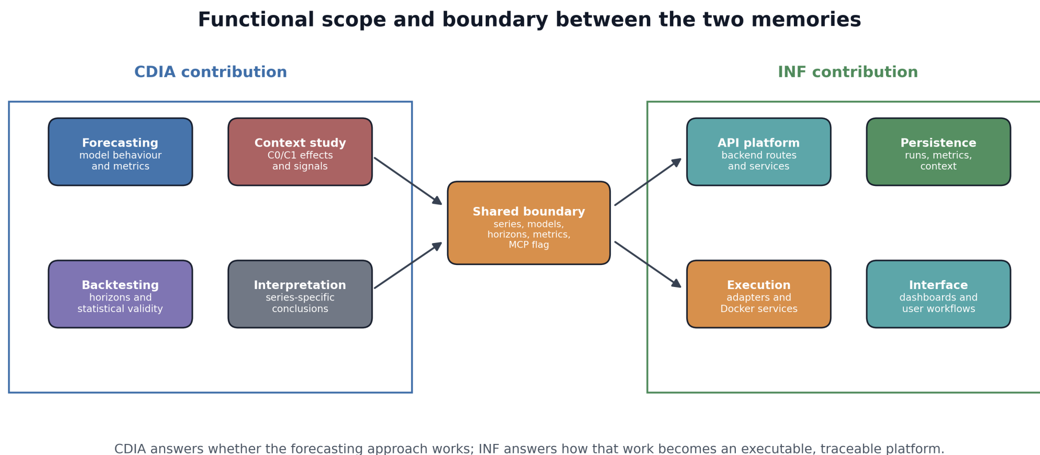


Figure 3.1.: Functional scope of the INF platform and its relation to the forecasting workflow.

tion logic for the platform entities, drift analysis, news context, and simulation. It also coordinates the execution of forecasting adapters and the retrieval of contextual information when MCP is enabled.

The frontend is responsible for the user-facing part of the platform. Its scope includes authentication screens, experiment lists, experiment detail views, run inspection, model comparison dashboards, simulation views, and contextual-information visualizations. The frontend does not implement forecasting models. It consumes the backend API and presents the results in a way that makes the system usable.

The persistence scope includes PostgreSQL and MongoDB. PostgreSQL stores structured platform entities, while MongoDB supports contextual and news-related documents used by the MCP server. This separation reflects the different nature of the data handled by the platform.

The infrastructure scope includes Docker Compose and the gateway configuration. The platform is designed as a multi-service environment that can be executed together. This supports reproducibility, testing, and later extension.

3.5. SCOPE OF THE INF CONTRIBUTION

The INF contribution is limited to the software-engineering side of the integrated project. It includes the design and implementation of the platform, the organization of services, the integration of forecasting execution, the connection with MCP context, the persistence of experiments and results, the frontend workflows, the deploy-

ment configuration, and the basic validation mechanisms.

The project uses the forecasting work as its domain and as its main source of functional needs. The platform is connected to a modelling study on inflation forecasting with classical models, deep learning models, foundation models, and contextual signals. In this thesis, those models are treated from the point of view of integration.

The scope also includes maintainability and extensibility. The platform should make it possible to add new forecasting adapters, new contextual signal sources, new views, or new validation tools without redesigning the full system. This is why the thesis gives importance to architecture, service boundaries, API design, persistence, and deployment.

3.6. OUT OF SCOPE

Several elements are intentionally outside the scope of the INF contribution.

First, the INF thesis does not aim to prove which forecasting model is best. It describes how model families are exposed through the platform; the detailed experimental comparison is outside its scope.

Second, the project does not develop new forecasting algorithms from scratch as part of the INF contribution. The software work integrates existing model logic and exposes it through a platform architecture. Improvements to the mathematical modelling itself are part of the data-science scope.

Third, the platform is not presented as a production banking or public-sector forecasting system. It is a final-degree engineering platform that demonstrates the architecture, integration, deployment, and user workflows of the project. A production deployment would require stronger security, access control, monitoring, auditability, data governance, scalability testing, and operational procedures.

Fourth, the current drift and monitoring capabilities are intentionally limited. The platform includes a first mechanism for residual-distribution comparison, but it does not implement a full monitoring, alerting, retraining, or model-governance pipeline. These aspects are suitable future work.

Fifth, the MCP integration is treated as a software capability for retrieving contextual information. The INF thesis does not claim that every MCP-derived signal improves forecasts.

3.7. EXPECTED CONTRIBUTION

The expected contribution of the INF part is a working software architecture that gives operational form to the forecasting study. It connects research results with workflows for creating experiments, executing runs, storing outputs, comparing metrics, and inspecting contextual information.

The expected result is an engineering base with modularity, service separation, reproducible deployment, API-based access, contextual integration, persistent experiment management, and user-facing visualization. These elements support validation and continuation after the current thesis is finished.

4. PLANNING

4.1. PLANNING APPROACH

This chapter presents the global planning of the integrated PFG. The project combines two differentiated contributions: the CDIA work, focused on the investigation and evaluation of inflation forecasting models, and the INF work, focused on the later development of the software platform that organises and exposes that forecasting workflow.

The planning follows an Agile and Kanban-oriented approach. The project was not managed as a fixed linear sequence where every detail was known from the beginning. Instead, the work was divided into user stories, tasks, and viable milestones. This made it possible to start with the CDIA investigation, check the forecasting results, and then move into the INF architecture and platform development with a clearer understanding of what the system had to support.

The complete workload is estimated at 725 hours. Of these, 365 hours correspond to the CDIA profile and 360 hours correspond to the INF profile. The CDIA effort is concentrated mainly in the first part of the project, where the research, data preparation, modelling, contextual-signal analysis, and evaluation were carried out. The INF effort becomes dominant afterwards, once the results had been checked and the platform requirements could be derived from the actual forecasting workflow. The writing of the CDIA memory overlapped with the INF development phase.

4.2. TASK DEFINITION

The work was organised into user stories so that each part of the project had a clear purpose and a measurable output. Table 4.1 summarizes the main user stories, their degree profile, and the tasks associated with each one.

Table 4.1.: User stories and main tasks of the integrated project

User story	Profile	Main tasks
US-1 Define the forecasting problem	CDIA	Identify the economic problem, define Spain CPI, Global CPI and European HICP as targets, formulate the research question, and decide the main model families to compare.
US-2 Prepare the data foundation	CDIA	Search data sources, clean and align monthly series, review candidate macroeconomic and institutional variables, and prepare the datasets for experimentation.
US-3 Build the initial forecasting workflow	CDIA	Use Spain CPI as the pilot case, execute exploratory analysis, define the rolling-origin protocol, and test classical baselines and initial foundation-model runs.
US-4 Add contextual conditions	CDIA	Define `C0`, `C1_inst`, `C1_mcp`, and `C1_full`; construct institutional, macroeconomic, energy, uncertainty, and MCP/text-derived signals.
US-5 Complete cross-series evaluation	CDIA	Extend the experiments to Global CPI and European HICP, compare model families by horizon, compute metrics, review results, and interpret the differences across targets.
US-6 Translate results into platform requirements	Shared	Convert datasets, models, contexts, runs, predictions, metrics, and results into software concepts that the INF platform can represent.
US-7 Implement the backend and persistence layer	INF	Design the API, domain entities, authentication, PostgreSQL persistence, MongoDB context storage, experiment workflow, predictions, and metrics.
US-8 Integrate forecasting and MCP services	INF	Build model adapters, connect MCP contextual access, configure MLflow tracking, and expose the forecasting workflow through platform services.
US-9 Build user workflows and validation	INF	Implement frontend views, comparison dashboards, forecast visualisations, simulation screens, Docker Compose deployment, health checks, tests, and drift analysis.
US-10 Prepare the final documentation	Shared	Write and review the CDIA and INF memories, prepare figures and tables, align shared sections, complete appendices, and collect definitions and bibliography.

4.2.1. General tasks

Some user stories are general because they connect both degree profiles. US-6 is shared because the INF platform requirements come directly from the CDIA experimental outputs. US-10 is also shared because the two memories must remain coherent while still presenting separate degree-specific contributions.

4.2.2. CDIA-specific tasks

The CDIA-specific work is represented by `US-1` to `US-5`. These tasks cover the investigation and analysis part of the project: economic problem definition, data source selection, dataset preparation, model comparison, contextual-signal design, rolling-origin evaluation, and interpretation of the results.

4.2.3. INF-specific tasks

The INF-specific work is represented by `US-7` to `US-9`. These tasks were developed after the main CDIA results had been checked. They cover the backend, persistence layer, forecasting adapters, MCP integration, frontend workflows, Docker environment, validation, and maintainability of the platform.

4.3. MILESTONES

The project was divided into three viable milestones. These milestones are used in the same sense as MVPs in Agile environments: each one marks a point where the project has reached a coherent and reviewable state.

1. `MVP-1`**: Forecasting evidence base.** This milestone includes `US-1` to `US-5`. At this point, the CDIA investigation is complete enough to explain the inflation forecasting problem, the selected datasets, the model families, the contextual conditions, and the main empirical results.

2. `MVP-2`**: Operational forecasting platform.** This milestone includes `US-6` to `US-9`. At this point, the forecasting workflow has been translated into platform requirements and implemented through backend services, persistence, adapters, MCP integration, frontend views, deployment, and validation checks.

3. `MVP-3`**: Final integrated documentation.** This milestone includes `US-10`. At this point, the two memories are written, reviewed, and aligned with the integrated nature of the project, while preserving the specific contribution of each degree.

4.4. TASK DISTRIBUTION

Table 4.2 presents the estimated effort by phase. The distribution reflects the actual chronology of the work: the first part is mainly CDIA, the second part is mainly INF, and the final documentation period overlaps both memories.

Table 4.2.: Estimated workload distribution by phase

Phase	CDIA hours	INF hours	Main output
Initial topic definition and CDIA research framing	30	5	Research question and first integrated-project idea
Literature review and data source definition	70	0	State-of-the-art, target series, and candidate signals
Spain CPI pilot and contextual-signal exploration	115	0	First forecasting workflow and C0/C1 logic
Global CPI, European HICP, and final CDIA evaluation	95	0	Cross-series results and interpretation
Transition from CDIA results to INF requirements	10	55	Platform requirements derived from the forecasting workflow
Backend, persistence, and forecasting integration	5	105	API, data model, storage, runs, predictions, metrics, and adapters
MCP integration, frontend, deployment, and validation	5	130	Context access, user workflows, Docker environment, and checks
Memory writing and final alignment	35	65	CDIA and INF reports, figures, tables, appendices, and review
Total	**365**	**360**	**725 hours of integrated project work**

Figure 4.1 summarises the same sequence graphically: CDIA research, translation of results into software requirements, INF development, and the final documentation/review period.

4.5. HUMAN-RESOURCE PLAN

The project was carried out individually, but for planning purposes it is useful to separate the work into professional roles. This makes the responsibilities clearer and also provides the basis for the budget chapter. Table 4.3 presents the roles considered in the plan.

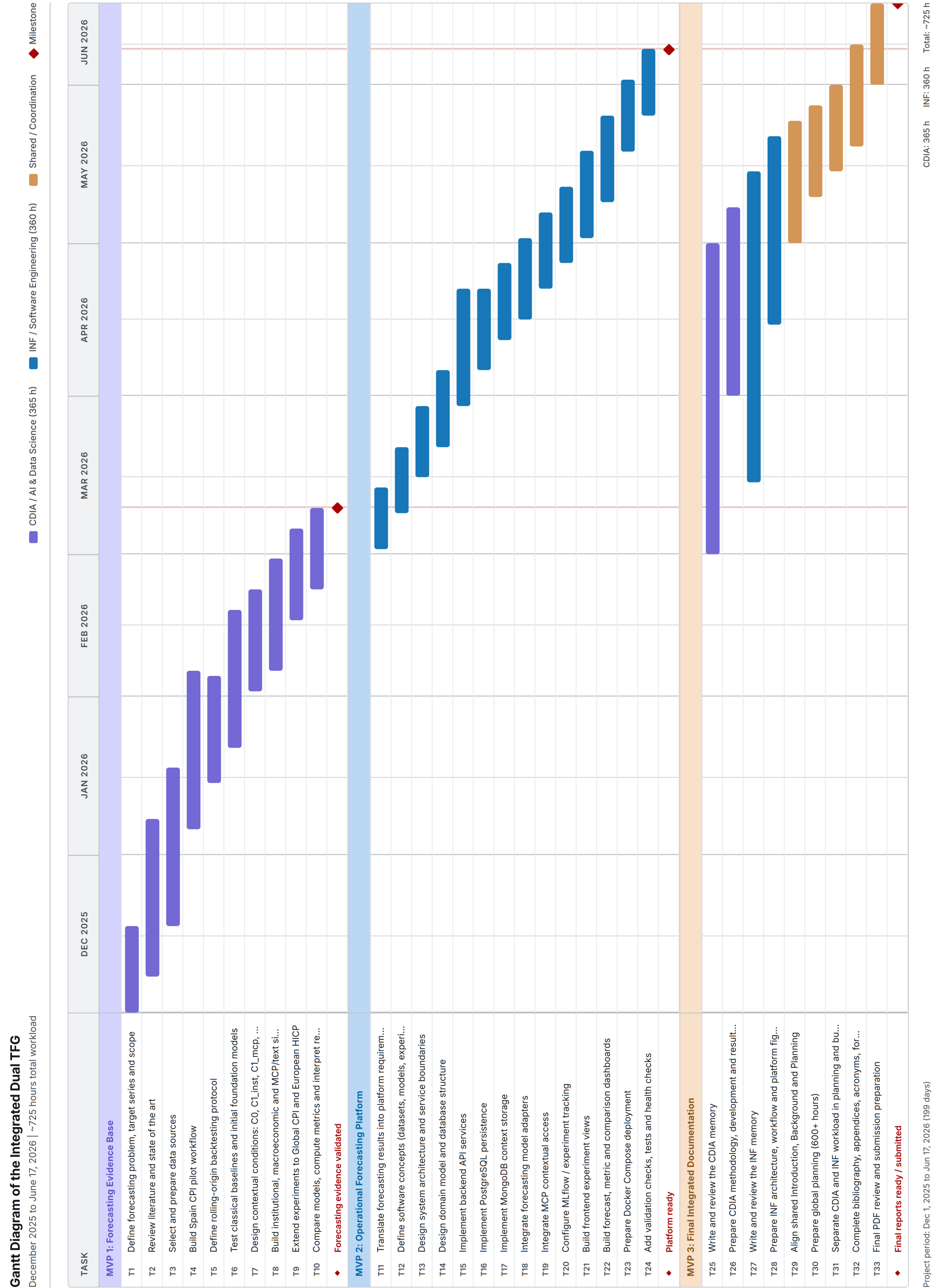


Figure 4.1.: Integrated Gantt diagram distinguishing MVPs, user stories, degree profiles, and estimated effort

Table 4.3.: Roles, responsibilities, and assigned personnel

Role	Responsibilities	Assigned person
Project director	Academic supervision, scope guidance, milestone review, and final validation of the project.	Supervisor
Project manager	Planning, Kanban-style task control, coordination between CDIA and INF, risk management, and final alignment.	Student
CDIA research analyst	State-of-the-art review, forecasting problem framing, and definition of the research gap.	Student
CDIA data scientist	Data source review, preprocessing, exploratory analysis, feature construction, and contextual-signal preparation.	Student
Machine-learning specialist	Model configuration, statistical and neural baselines, foundation-model experiments, rolling-origin evaluation, and result analysis.	Student
Software architect	Requirements analysis, architecture design, service boundaries, technology decisions, and platform structure.	Student
Backend and integration engineer	API implementation, persistence, forecasting adapters, MCP integration, MLflow, Docker Compose, and validation checks.	Student
Frontend engineer	User interface, experiment views, comparison dashboards, forecast visualisation, contextual views, and simulation screens.	Student
Technical writer	CDIA and INF memory writing, figures, tables, appendices, bibliography, and final review.	Student

Table 4.4 shows the estimated workload by role. The project director hours are shown separately because they represent supervision effort and are not included in the 725 hours of direct student work.

Table 4.4.: Estimated workload distribution by role

Role	Percentage of student work	Hours
Project director	Additional supervision	30
Project manager	3.3%	24
CDIA research analyst	6.9%	50
CDIA data scientist	12.4%	90
Machine-learning specialist	17.2%	125
CDIA evaluation and technical writing	13.8%	100
Software architect	6.9%	50
Backend and integration engineer	23.4%	170
Frontend engineer	8.3%	60
INF validation and documentation	7.7%	56
Total student workload	**100%**	**725**

4.6. PLANNING RISKS

The first planning risk was the uncertainty of the CDIA investigation. At the beginning of the project, it was not possible to know which model family would perform best or whether contextual signals would improve the forecasts. This was managed by starting with Spain CPI as a pilot case before extending the workflow to Global CPI and European HICP.

A second risk appeared during the transition from CDIA to INF. The platform could only be designed properly once the forecasting workflow and result structure were clear enough. This was managed by defining general platform concepts such as datasets, models, experiments, runs, predictions, metrics, and contexts.

A third risk was integration complexity. The INF block combined backend services, frontend screens, databases, MCP tools, MLflow, a gateway, and Docker Compose. This was managed by keeping clear service boundaries and by treating deployment and validation as explicit tasks rather than final additions.

5. BUDGET

With the common planning of the integrated project defined, this chapter estimates the economic cost of the work. The budget is not intended to represent real money paid during the project, but a professional valuation of the human effort, supervision, equipment, and technical resources that would be needed to develop an equivalent project.

The budget follows the workload distribution described in the previous chapter. It includes the complete integrated project, not only the CDIA part, because the requirements of a dual PFG ask for a global budget with the allocation of each profile clearly identified.

5.1. HUMAN RESOURCES

The main cost of the project is human work. Although the project was developed individually, the workload has been separated into professional roles so that the estimate is clearer. The project director is included as academic supervision, while the remaining roles correspond to the direct work carried out in the integrated project.

Table 5.1.: Estimated salary expenses

Human resource	**Unit cost**	**Hours**	**Total**
Project director	50 EUR/h	30	1,500 EUR
Project manager	35 EUR/h	24	840 EUR
CDIA research analyst	25 EUR/h	50	1,250 EUR
CDIA data scientist	30 EUR/h	90	2,700 EUR
Machine-learning specialist	35 EUR/h	125	4,375 EUR
CDIA evaluation analyst	30 EUR/h	55	1,650 EUR
CDIA technical writer	22 EUR/h	45	990 EUR
Software architect	35 EUR/h	50	1,750 EUR
Backend and data engineer	32 EUR/h	85	2,720 EUR
Integration and DevOps engineer	34 EUR/h	85	2,890 EUR
Frontend engineer	28 EUR/h	60	1,680 EUR
Validation and documentation engineer	26 EUR/h	56	1,456 EUR
Subtotal human resources	–	–	**23,801 EUR**

Table 5.2 separates the human-resource cost by project profile. The shared coordination and supervision costs

are kept separate because they support both memories.

Table 5.2.: Human-resource cost by project profile

Profile	Direct hours	Cost
Shared coordination and supervision	54	2,340 EUR
CDIA-specific work	365	10,965 EUR
INF-specific work	336	10,496 EUR
Total	**755**	**23,801 EUR**

The 755 hours in Table 5.2 include 30 hours of academic supervision. The direct student workload remains 725 hours, as defined in the planning chapter.

5.2. MATERIALS AND EQUIPMENT

The project was developed with personal equipment and mainly open-source software. However, an equivalent professional budget should include the proportional cost of the hardware and technical resources required to run experiments, develop the platform, store results, and maintain the documentation.

Table 5.3.: Estimated materials and equipment costs

Item	Cost	Justification
Personal workstation amortisation	450 EUR	Proportional use of the development computer during the project
External model and API access	250 EUR	Reference allowance for TimeGPT and occasional external model calls
Local execution and electricity	120 EUR	Repeated model executions, backend/frontend runs, and Docker Compose testing
Storage and backups	80 EUR	Datasets, model outputs, figures, PDFs, repositories, and memory files
Connectivity and workspace infrastructure	150 EUR	Internet connection and general project infrastructure
Software licences	0 EUR	Main tools and libraries were open-source or available at no additional cost
Datasets	0 EUR	The data sources used were public or freely accessible
Subtotal materials and equipment	**1,050 EUR**	–

5.3. TOTAL BUDGET

The total budget is obtained by adding human resources and material resources. Table 5.4 summarizes the final estimate.

Table 5.4.: Budget summary

Concept	**Total cost**
Human resources	23,801 EUR
Materials and equipment	1,050 EUR
Total estimated budget	**24,851 EUR**

5.4. INTERPRETATION

The budget shows that the project is mainly a knowledge-work project. The largest cost is the time required to investigate the forecasting problem, prepare data, compare models, interpret results, design the platform, implement the software architecture, validate the system, and document both memories.

Material costs are limited because the project relies mostly on public data, open-source tools, and local development resources. This makes the project feasible from an economic point of view while still giving it the structure of a professional integrated project.

6. METHODOLOGY

6.1. METHODOLOGICAL APPROACH

The methodology of the INF contribution follows an incremental software-engineering approach. It explains how the forecasting work was transformed into an organized software platform. The detailed implementation of the backend, frontend, persistence layer, MCP integration, deployment environment, and validation mechanisms is reserved for Chapter 7.

The work was organized around a practical sequence: requirements analysis, architectural design, iterative implementation, integration, validation, and documentation. These phases were not treated as isolated blocks. Several decisions were revised when the forecasting workflow became clearer, when repository constraints appeared, or when the platform needed to expose a capability in a more maintainable way. The software system depends on data-science artifacts, external services, and evolving experiment outputs, so the methodology kept room for iteration.

The methodological criterion used throughout the INF work was to identify the software needs created by the forecasting workflow and turn them into platform responsibilities.

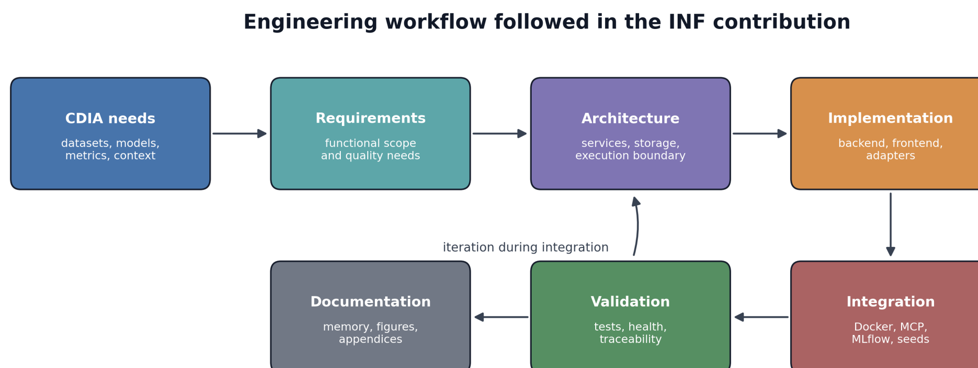


Figure 6.1.: Methodological workflow followed in the INF contribution.

Figure 6.1 summarizes the sequence followed during the project. The workflow starts from forecasting needs

and moves through software requirements, architecture, implementation, integration, validation, and documentation. The first part of this chapter develops the analysis and design phases, because they define the basis for the development work described later.

6.2. CONNECTION WITH THE FORECASTING WORKFLOW

The starting point of the methodology was the forecasting workflow. That workflow defines the domain of the system: inflation series, contextual economic signals, forecasting model families, experiment conditions, forecast horizons, and evaluation outputs. These elements were analysed as software entities and system responsibilities.

This connection shaped the platform requirements. A dataset had to be registered, described, selected, and related to future runs. A model had to be represented as a catalogue entry linked to the adapter layer. A forecast run had to preserve its input configuration, predictions, metrics, timestamps, and status. Contextual information had to be requested, stored, and attached to user workflows.

The methodology treated the forecasting workflow as the source of functional needs. The platform does not decide whether a model is scientifically better than another one; it provides the structure needed to execute, inspect, compare, and extend forecasting work in a controlled way.

This relationship also influenced the order of the work. Before designing screens or database entities, it was necessary to understand which artifacts had to be represented in the platform. The analysis considered the project outputs that a user would reasonably need to access and later mapped them to backend resources, frontend views, and the persistence model.

6.3. REQUIREMENTS ANALYSIS

The first methodological phase was requirements analysis. The objective was to translate the needs of the integrated forecasting project into software requirements that could guide the implementation. This analysis was based on the repository structure, the forecasting experiment workflow, the expected user interaction with the system, and the maintainability needs of the platform.

The functional requirements were grouped around the main workflows of the platform. The system had to support user access, dataset management, model catalogue inspection, experiment creation, forecast-run execution, metric storage, prediction visualization, contextual-information retrieval, drift analysis, and simulation.

These requirements were selected because they correspond to the operations needed to make the forecasting study usable through a platform.

The analysis also identified non-functional requirements. The platform had to be modular, reproducible, extensible, and understandable. These qualities were necessary because the project depends on several services, external configuration, forecasting adapters, contextual signals, and user-facing workflows that may evolve after the thesis is completed.

Another requirement was traceability. In a forecasting platform, a result must remain connected to the configuration that produced it. The methodology treated that relationship as a design constraint for the backend entities, the database model, and the way results are exposed to the frontend.

6.4. ARCHITECTURE DESIGN METHOD

After defining the requirements, the next phase was the design of the system architecture. The architecture was designed by decomposing the platform into components with clear responsibilities. This decision follows from the nature of the project: the system must coordinate user interaction, API logic, forecasting execution, contextual signal retrieval, persistent storage, experiment tracking, and deployment.

The first design decision was to separate backend and frontend responsibilities. The backend was defined as the service responsible for application logic, data access, forecasting orchestration, MCP communication, authentication-related workflows, metrics, and API exposure. The frontend was defined as the user-facing layer responsible for presenting experiments, runs, predictions, comparisons, simulations, and contextual information. This separation gives each side a clear boundary for implementation and future extension.

The second design decision was to represent forecasting functionality through adapters. The forecasting work includes different model families and execution paths, but the platform needs a common way to request and expose forecasts. The adapter idea reduces coupling between the API layer and the internal implementation of each model. From the methodology perspective, this was treated as a design safeguard: changes in model code should not require a complete redesign of the platform.

The third design decision was to separate storage responsibilities. Structured platform entities were assigned to a relational database model. Contextual and news-related information was treated as semi-structured data, more naturally connected with document storage and MCP retrieval. This division reflects the different shape of the information handled by the platform and supports future extension.

The fourth design decision was to make deployment part of the architecture instead of treating it as a final accessory. The platform depends on several services, so reproducible execution is part of the engineering problem. For this reason, the methodology included the design of a multi-service environment from the beginning, with backend, frontend, gateway, databases, experiment tracking, and MCP service considered as connected pieces of the same system.

These design decisions define the architectural baseline used in the development chapter.

6.5. ITERATIVE IMPLEMENTATION

The implementation phase followed an iterative strategy. Instead of building the complete platform in a single block, the work was divided into functional increments. Each increment added a coherent part of the system and made it possible to check whether the architecture still matched the needs of the forecasting project.

The first implementation increments focused on the backend foundations. This included the FastAPI application structure, configuration, database access, domain entities, schemas, and the first API endpoints. The objective of this stage was to establish the core application model before connecting more complex forecasting or contextual functionality. Working in this order kept the interface aligned with stable platform concepts.

The next increments introduced forecasting execution through adapters. This phase connected the platform with the model families used in the project while preserving a common interaction pattern. Each adapter had to receive a request in a comparable format, execute or represent the corresponding forecasting logic, and return outputs that the rest of the platform could store and display. This allowed models to be added progressively while keeping the API independent from the internal details of each forecasting implementation.

The frontend was implemented in parallel with the backend once the main API resources were stable enough. The interface was developed around the user workflows defined during requirements analysis. These workflows include dataset and experiment access, run inspection, prediction charts, metric views, model comparison, context views, and simulation support. This avoided treating the frontend as a decorative layer. Its role in the methodology was to verify whether the backend concepts were understandable and useful from the user's point of view.

The implementation method also included regular refactoring. As the project incorporated new capabilities, such as MCP context, drift analysis, MLflow logging, or ensemble-stack execution, some earlier assumptions had to be adjusted. The work had to make each feature operate while keeping the repository organized enough

for the system to remain explainable in the thesis and extensible in future work.

6.6. INTEGRATION STRATEGY

Integration was one of the central methodological concerns of the INF contribution. The platform is useful only if its parts work together: backend services, frontend screens, forecasting artifacts, MCP context, databases, experiment tracking, and deployment configuration. For this reason, integration was treated as a continuous activity rather than as a final step after implementation.

The first integration axis was the connection between the INF platform and the forecasting repository. The platform needed access to processed datasets, model outputs, and shared utilities without mixing platform code with research scripts. The methodology kept the forecasting code as the analytical base and used the INF platform layer to expose, organize, and inspect its outputs.

The second integration axis was the connection with MCP and contextual information. MCP was treated as an external service capability. The backend had to request macroeconomic or news-related signals, handle possible service errors, and attach the returned information to forecasting or visualization workflows. Contextual data can be incomplete, delayed, or unavailable, so the integration method included graceful handling of missing information and a clear separation between retrieving context and interpreting its predictive value.

The third integration axis was persistence. The methodology connected application workflows with PostgreSQL for structured entities and MongoDB for news or contextual records. This ensured that platform outputs were not treated as temporary files. Persisting these elements supports traceability and makes the platform closer to a real engineering system.

The fourth integration axis was service orchestration. Docker Compose was used to check that the multi-service stack could be executed as a connected environment. This execution model reduces dependence on manual local setup and supports platform-level evaluation.

6.7. VALIDATION STRATEGY

The validation strategy was designed according to the scope defined in Chapter 3. The objective was to validate that the software platform behaves coherently: endpoints respond as expected, data entities are stored and retrieved correctly, services can communicate, and the user-facing workflows expose the relevant information.

Validation was approached at several levels. At the backend level, the project uses tests for the main API workflow groups. These tests help verify that the main resources of the platform are accessible and that changes in one part of the backend do not silently break another part.

At the integration level, validation focused on the interaction between services. This included checking that the backend can access databases, that MCP communication can be requested through the configured client, that run outputs can be stored and retrieved, and that the containerized environment connects the required services. The Docker Compose configuration was part of the validation method, not only a deployment artifact.

At the user-workflow level, validation was based on the consistency between the frontend and the backend. The interface had to present the platform entities and contextual views in a way that matched the API responses. A technically correct endpoint can still fail as a platform feature if the user cannot inspect or understand the result.

The platform also includes initial observability and monitoring-oriented mechanisms. MLflow logging supports experiment tracking, and the drift endpoint provides a first residual-distribution comparison based on a Kolmogorov-Smirnov test. These mechanisms show that the architecture can support traceability and later monitoring extensions.

6.8. DOCUMENTATION AND REPRODUCIBILITY

Documentation was treated as part of the methodology because the project must be understandable as both a software system and a final-degree thesis. The repository structure, service responsibilities, configuration, endpoints, persistence model, and deployment process had to be clear enough for another developer or evaluator to understand how the platform is organized.

Reproducibility was addressed through several decisions. The repository keeps the forecasting and architecture areas in a single project structure, shared utilities are separated from service-specific code, configuration is externalized through environment variables, and the main services are coordinated with Docker Compose. This does not remove all reproducibility challenges, because some external APIs, m

7. DEVELOPMENT

7.1. PURPOSE OF THE CHAPTER

This chapter presents the development of the INF contribution: the software platform that supports the inflation forecasting project. It moves from the overall system design to the implementation modules that make the platform executable, persistent, and usable.

As defined in Chapter 1, this chapter focuses on how the platform entities and user workflows are organized through a coherent software system.

The development work is centred on the INF platform implementation under `tfg-arquitectura`, while keeping a controlled connection with `tfg-forecasting` and `shared`. The INF platform includes the multi-service stack, forecasting adapters, and validation tests.

7.2. DEVELOPMENT SCOPE

The platform was developed as an engineering layer around an existing forecasting research workflow. It transforms forecasting assets into an executable and inspectable system, with the concrete entity model developed in the backend and persistence sections below.

The scope of the development covers the following technical blocks:

- backend API for application workflows and forecasting execution;
- relational persistence for datasets, users, models, experiments, runs, predictions, and metrics;
- document-oriented storage for contextual and news-related information;
- forecasting adapter registry for the model families exposed by the platform;
- MCP service integration for macroeconomic signals, news context, and sentiment information;
- frontend interface for experiment management, result inspection, model comparison, simulations, and contextual visualizations;

- Docker Compose environment for running the platform as a connected set of services;
- experiment tracking and first monitoring mechanisms through MLflow and drift analysis;
- backend tests for the main API workflows.

The development scope excludes the design of new forecasting algorithms and full model-performance evaluation. The INF goal is to demonstrate a modular, reproducible, and extensible architecture around the forecasting work.

7.3. REQUIREMENTS IMPLEMENTED

The requirements were derived from the objectives and methodology defined in the previous chapters. They are presented here because they guide the technical development that follows. Functional requirements describe what the system must do. Non-functional requirements describe the quality constraints that shape the design.

7.3.1. Functional requirements

Table 7.1 summarizes the main functional requirements implemented by the platform.

Table 7.1.: Functional requirements of the INF platform

ID	Requirement	Purpose in the platform
FR-01	User access and roles	Allow users to access the platform and support role-based administration workflows.
FR-02	Dataset and series catalogue	Register the inflation datasets and their target series so experiments can be created from controlled sources.
FR-03	Model catalogue	Expose the available model families through stable catalogue entries and active/inactive status.
FR-04	Experiment management	Allow experiments to be created, listed, inspected, updated, and connected to a target series, model, horizon, and configuration.
FR-05	Forecast run execution	Trigger forecast runs and store their execution status, timestamps, errors, predictions, and metrics.
FR-06	Prediction and metric storage	Persist forecast outputs so they can be inspected, compared, and reused without rerunning every model.
FR-07	Model comparison	Provide backend and frontend support for comparing model performance across experiments and horizons.
FR-08	MCP context retrieval	Retrieve contextual macroeconomic signals and news sentiment through the MCP service when required by a workflow.
FR-09	News and context visualization	Present contextual information in the frontend so forecast results are not isolated from their economic context.
FR-10	Simulation support	Provide what-if or simulation workflows that help users inspect possible forecast behaviour under selected assumptions.
FR-11	Drift analysis	Include a first endpoint for residual-distribution drift detection as a monitoring-oriented extension.
FR-12	Health and operational checks	Expose health information so the service state can be verified during development and deployment.

These requirements define the minimum functional surface of the platform: application entities, execution workflows, persistent outputs, context retrieval, and a reproducible service environment.

7.3.2. Non-functional requirements

Table 7.2 summarizes the non-functional requirements used to guide the architecture.

Table 7.2.: Non-functional requirements of the INF platform

ID	Requirement	Design implication
NFR-01	Modularity	Separate backend, frontend, forecasting adapters, persistence, MCP service, gateway, and infrastructure.
NFR-02	Reproducibility	Use explicit configuration, repository organization, Docker Compose, seeded data, and persisted outputs.
NFR-03	Extensibility	Allow new datasets, models, adapters, contextual signals, visualizations, and monitoring mechanisms to be added later.
NFR-04	Traceability	Preserve the relation between platform entities, contextual inputs, and timestamps.
NFR-05	Maintainability	Keep responsibilities clear enough for the system to be understood, tested, documented, and continued after the TFG.
NFR-06	Robustness to external services	Handle unavailable MCP, API, or model services without making the whole platform unusable.
NFR-07	Usability	Present forecasts, comparisons, metrics, and context through inspectable workflows.
NFR-08	Validation support	Include tests, health checks, metric storage, and initial observability mechanisms.

The central non-functional requirement is traceability. Each forecasting result must remain connected to its dataset, model, horizon, configuration, run, predictions, metrics, and contextual information. This requirement

influenced the database model, the API design, and the frontend views.

7.4. REPOSITORY ORGANIZATION

The repository is organized as a monorepo with two main project areas and one shared support layer. This structure lets the platform expose, store, and visualize artifacts from the forecasting pipeline while keeping the software architecture in a separate area.

Table 7.3.: Repository areas used by the INF platform

Area	**Role**	**Use in the INF development**
`tfg-arquitectura/`	Software platform	Contains the backend, frontend, gateway, databases, infrastructure files, MCP server, migrations, and tests.
`tfg-forecasting/`	Forecasting pipeline	Contains datasets, ETL scripts, model implementations, evaluation outputs, processed Parquet files, and result artifacts used by the platform.
`shared/`	Common utilities	Provides reusable constants, data-loading helpers, metric functions, and logging utilities shared by forecasting and architecture code.
`docker-compose.yml`	Service orchestration	Defines the executable multi-service environment.
`PROJECT_CONTEXT.md`	Project context document	Summarizes the current project status, architecture, datasets, model families, results, and important technical decisions.

Inside `tfg-arquitectura/`, the backend is implemented in `backend/`, the frontend in `frontend/`, the reverse proxy in `gateway/`, the database initialization scripts in `db/`, and the MCP service in `mcp_server/`. This separation allows each part of the platform to be explained and evolved independently.

The connection with the forecasting area is controlled through mounted volumes, shared utilities, and result

artifacts. The platform uses the forecasting repository as the analytical base and builds a service layer around the elements that need to be exposed: datasets, model catalogue entries, forecast executions, predictions, metrics, and contextual signals.

This organization also supports reproducibility. The same repository contains the research pipeline, the platform implementation, shared utilities, and service orchestration, so the scientific source of the forecasts and the architecture that exposes them can be inspected together.

7.5. GENERAL ARCHITECTURE

The implemented system follows a service-oriented architecture. The backend exposes the main application workflows through a REST API, and the frontend consumes that API to provide the user-facing interface. Persistence, experiment tracking, contextual retrieval, gateway routing, and orchestration are separated across the multi-service stack.

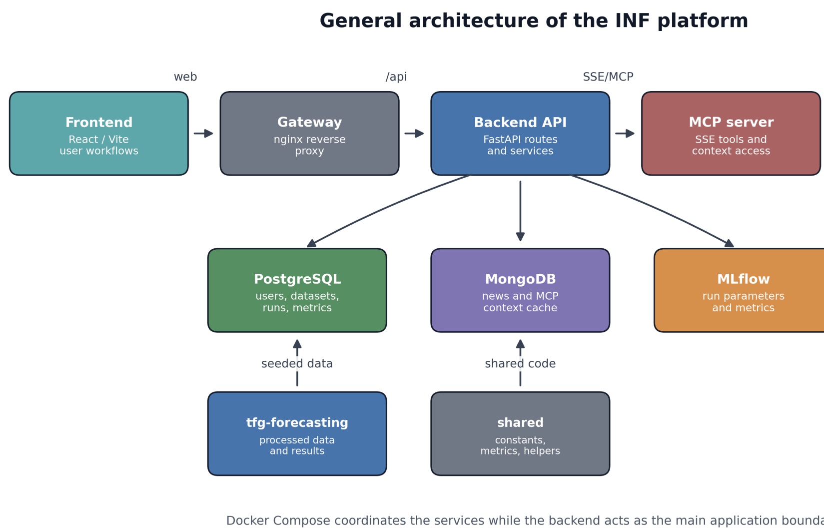


Figure 7.1.: General architecture of the INF platform.

Figure 7.1 summarizes the main components and their relationships across the user interface, backend API, persistence, tracking, contextual retrieval, forecasting artifacts, shared utilities, and the Docker Compose execution boundary.

Table 7.4.: Main components of the developed platform

Component	**Repository area**	**Main responsibility**
Backend API	`backend`	Exposes authentication, users, datasets, experiments, runs, metrics, drift, news, assistant, and simulation workflows.
Frontend application	`frontend`	Provides the user interface for creating experiments, inspecting runs, comparing models, viewing forecasts, exploring context, and using simulations.
Forecasting adapters	`backend/app/forecasting`	Provide a common execution interface for naive, ARIMA, AutoARIMA, SARIMA, SARIMAX, Ridge, TimesFM, Chronos-2, TimeGPT, and ensemble-stack models.
Relational database	PostgreSQL service	Stores structured entities such as users, datasets, series, observations, model catalogue entries, experiments, runs, predictions, and metrics.
Document database	MongoDB service	Stores or serves news and contextual documents used by the MCP and news-related workflows.
MCP server	`mcp_server`	Exposes macro signals, news context, and news sentiment tools through MCP over SSE transport.
Experiment tracking	MLflow service	Records run parameters and metrics as a first experiment-tracking layer.
Gateway	`gateway`	Provides the reverse-proxy entry point for the deployed web platform.
Forecasting repository	`tfg-forecasting`	Contains processed datasets, model scripts, evaluation outputs, and forecasting artifacts used as the analytical base of the platform.
Shared utilities	`shared`	Provides common constants, data loading, metrics, and logging support used across the project.

The architecture follows the scope defined earlier in the thesis. The forecasting repository remains the analytical source of models, data, and results, while the architecture layer turns those elements into services, entities, workflows, and visual interfaces.

7.6. BACKEND AND API

The backend is the central coordination layer of the platform. It exposes the application workflows through a REST API, connects with the databases, launches forecast runs, retrieves contextual information when MCP is enabled, stores predictions and metrics, and provides the data consumed by the frontend. It is implemented with FastAPI and organized as a modular application under `tfg-arquitectura/backend` [17].

7.6.1. Application entry point

The main backend application is defined in `backend/app/main.py`. This file creates the FastAPI application, configures the API metadata, registers middleware, attaches exception handlers, initializes the application lifecycle, and includes the versioned API router.

During startup, the backend checks the PostgreSQL connection and seeds an administrator user when needed. During shutdown, it disposes the PostgreSQL engine and closes the MongoDB client. This lifecycle management supports graceful failure when an external service is unavailable during local or containerized execution.

The API documentation is exposed through the FastAPI documentation routes. In the current implementation, Swagger documentation is available under `/api/docs` and ReDoc under `/api/redoc`. These routes allow endpoint inspection during development and validation without reading the source code directly.

7.6.2. Router organization

The API is organized under a versioned router with the prefix `/api/v1`. Each functional area has its own router module and is then included in the versioned router. This gives the backend a clear structure and avoids concentrating all endpoints in a single file.

Table 7.5.: Backend API route groups

Router	Main module	Responsibility
Health	`health.py`	Exposes platform status checks used during development and deployment.
Authentication	`auth.py`	Provides sign-up and sign-in workflows based on JWT authentication.
Users	`users.py`	Supports user and role-management operations, mainly for administrative use.
Datasets	`datasets.py`	Exposes datasets, series, observations, and active model catalogue entries.
Experiments	`experiments.py`	Manages experiment creation, listing, inspection, deletion, and access to experiment runs.
Runs	`runs.py`	Triggers forecast execution and exposes run status, predictions, metrics, MCP context, and narrative analysis.
Metrics	`metrics.py`	Provides comparison-oriented metric outputs across experiments and runs.
Drift	`drift.py`	Provides the first residual-distribution drift check for completed runs.
What-if	`whatif.py`	Supports simulation workflows for inspecting alternative forecast behaviour.
News	`news.py`	Exposes news and context-related information used by frontend views.
Assistant	`assistant.py`	Provides an assistant-oriented endpoint connected with generated explanations.

This route grouping follows the domain workflow of the application: authentication, dataset inspection, experiment creation, run execution, prediction and metric review, contextual inspection, comparison, drift checks, and simulation.

7.6.3. Experiment lifecycle

The experiment workflow is a central backend responsibility. An experiment records the forecasting configuration selected by the user: target series, model, horizon, MCP usage, and optional configuration parameters. The backend checks that the selected series and model exist before creating the experiment. It also enforces ownership rules: administrators can access all experiments, while non-admin users can only access their own.

The concrete creation endpoint is `POST /api/v1/experiments`, implemented in `backend/app/api/v1/experiments.py`. Its request body is defined by the `ExperimentCreate` schema: `name`, `series_id`, `model_id`, `horizon`, `use_mcp`, and optional `config`. The backend verifies that the selected `Series` and `ModelCatalog` rows exist, creates an `Experiment` with the authenticated user's identifier, and returns an `ExperimentOut` response with status `created`. The integration test `test_create_experiment_researcher` checks the `201 Created` response, the persisted name, the horizon value, the `created` status, and the authenticated `user_id`.

```
POST /api/v1/experiments
Authorization: Bearer <researcher-token>

{
  "name": "Test experiment",
  "series_id": <existing_series_id>,
  "model_id": <existing_model_catalogue_id>,
  "horizon": 12,
  "use_mcp": false
}

201 Created
{
  "id": <generated_experiment_id>,
  "user_id": <authenticated_researcher_id>,
  "name": "Test experiment",
  "series_id": <existing_series_id>,
  "model_id": <existing_model_catalogue_id>,
  "horizon": 12,
  "use_mcp": false,
  "config": null,
  "status": "created",
```

```
    "created_at": <database_timestamp>,
    "updated_at": <database_timestamp>
}
```

Pseudocode 7.1: Example experiment-creation request and response shape

Once an experiment exists, a run can be triggered. The run starts in a pending state and is then executed in a background task. This avoids blocking the HTTP request while the forecasting adapter is running. During execution, the backend loads the target series from PostgreSQL, selects the correct forecasting adapter from the registry, optionally loads exogenous or MCP-derived features, runs the model, computes metrics, stores predictions and metrics, and updates the run status.

The run lifecycle uses explicit states: `pending`, `running`, `done`, and `failed`. This gives the frontend a clear representation of long-running executions, failures, output artifacts, and later inspection.

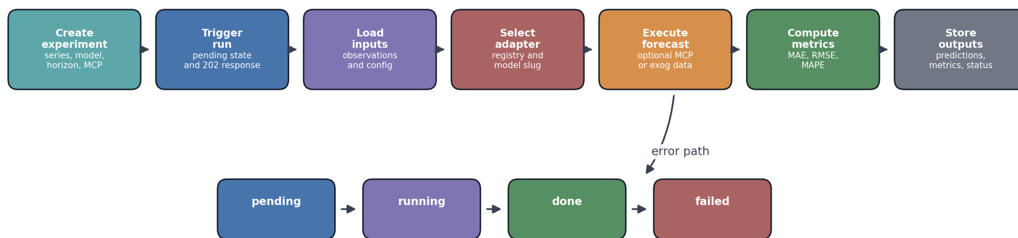


Figure 7.2.: Run execution flow and lifecycle states.

7.6.4. Authentication and permissions

The backend includes a basic user model with three roles: `admin`, `researcher`, and `viewer`. This separation is modest but useful for the scope of the project. It allows the platform to distinguish between administrative operations, experiment creation, and read-oriented access.

Authentication is based on signed tokens. Protected endpoints receive the current user through dependency

injection and apply role checks where required. Workflows such as experiment creation or deletion can be restricted, while read-oriented workflows can remain available according to the user's permissions.

This security layer demonstrates access-control design and prevents backend entities from being treated as anonymous unrestricted resources.

7.6.5. Assistant and narrative analysis

The platform includes two language-model-assisted explanation paths. The first one is the assistant router, implemented in `backend/app/api/v1/assistant.py`. It exposes `POST /api/v1/assistant/simulator/chat`, a protected streaming endpoint for the what-if simulator. The second one is the run-level narration endpoint, implemented in `backend/app/api/v1/runs.py` as `POST /api/v1/runs/{run_id}/narration`. This second endpoint is the one shown in the run-detail workflow as the optional "LLM Analysis" panel.

The simulator assistant uses the local Ollama chat API at `{OLLAMA_URL}/api/chat`. The default configuration in `backend/app/config.py` sets `OLLAMA_URL=http://host.docker.internal:11434` and `OLLAMA_CHAT_MODEL=llama3.2:3b`. The request body is a `SimChatRequest` with `messages` and optional `context`. Each message has `role` equal to `user` or `assistant` and a `content` string limited to 1500 characters. The optional simulator context contains `series_name`, `series_unit`, `horizon`, `signals`, `baseline`, `counterfactual`, `top_driver_label`, and `top_driver_contribution`. The endpoint keeps at most 30 messages, requires the last message to come from the user, injects the live simulator state into a system prompt, and streams plain UTF-8 text through `StreamingResponse`.

The run narration path uses a separate service in `backend/app/services/narration.py`. The endpoint first checks that the run exists, that the authenticated user owns it or is an administrator, and that the run status is `done`. It then reads the stored metrics, ordered prediction values, the experiment MCP flag, and, when MCP was enabled, the stored `mcp_contexts` document. These values are passed to `generate_narration` as `model_slug`, `metrics`, `predictions`, `use_mcp`, and optional `mcp_signals`.

```
POST /api/v1/runs/{run_id}/narration
Authorization: Bearer <user-token>

Inputs assembled by the backend:
{
  "model_slug": "<model_catalogue_slug>",
  "metrics": {"mae": ..., "rmse": ..., "mape": ...},
  "predictions": <ordered_prediction_values>,
  "use_mcp": <experiment_mcp_flag>,
  "mcp_signals": <stored_mcp_signal_rows_when_available>
}

Response:
{
  "narrative": "<generated_explanation_text>",
  "model": "llama3.2:3b"
}
```

Pseudocode 7.2: Inputs and output of the run narration endpoint

The narration service calls the local Ollama generate API at `{OLLAMA_URL}/api/generate`. The default model is `OLLAMA_MODEL=llama3.2:3b`; the request uses `stream=false`, `temperature=0.3`, and `num_predict=350`. The prompt starts with the fixed role instruction “You are a concise inflation analyst writing for a research report.” It then inserts the model slug, whether MCP macro context was used, the first forecast values, and the MAE, RMSE, and MAPE values. If MCP signals are available, it may also include the first available ECB and FOMC hawkishness scores. The final instruction asks for a professional three- or four-sentence analysis of the forecast trajectory and model accuracy, without bullet points or headers.

The feature is bounded as an explanation layer. It does not call a forecasting adapter, does not modify predictions, does not recalculate metrics, does not write corrected values back to PostgreSQL, and does not decide whether one model is scientifically better than another. If Ollama is unavailable, the endpoint returns `503 Service Unavailable`; the run remains valid because predictions and metrics have already been stored. This boundary also preserves the project split: MCP retrieves and stores context as a software capability, while the narrative service only summarizes already-stored outputs. The scientific interpretation of whether MCP or a model family improves forecasting performance remains part of the CDIA evaluation, not an inference made by

the assistant.

7.7. DATA MODEL AND PERSISTENCE

Persistence is one of the main differences between the platform and a collection of forecasting scripts. The platform stores application entities, user decisions, run outputs, prediction values, metrics, and contextual information so that results can be inspected later and connected to their origin.

The system uses two persistence technologies. PostgreSQL stores structured relational entities, and MongoDB stores contextual and news-related documents. This separation matches the nature of the data handled by the platform: experiments, runs, predictions, and metrics have clear relational structure, while news context and MCP outputs are more flexible documents.

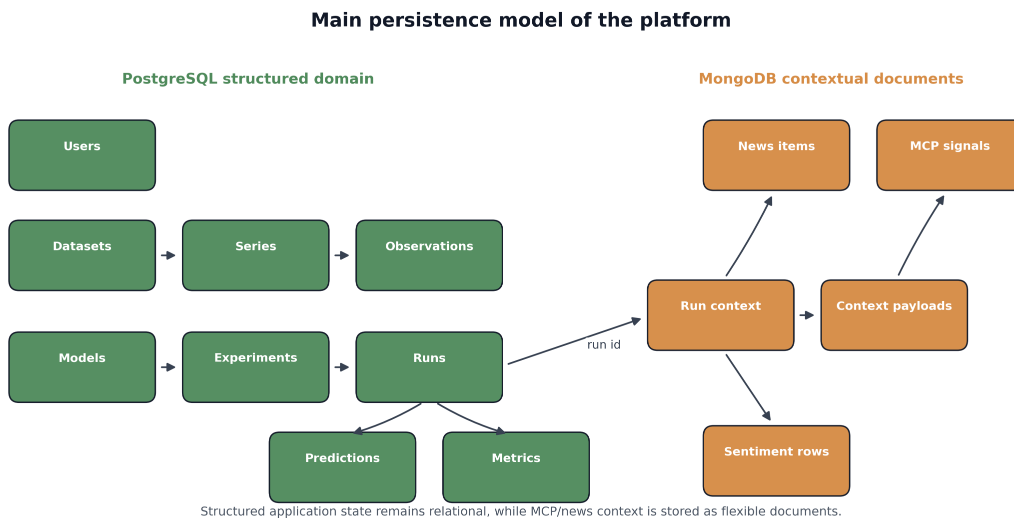


Figure 7.3.: Main persistence model of the INF platform.

Figure 7.3 shows the relation between the main PostgreSQL entities and the contextual MongoDB collections used by the MCP workflows.

7.7.1. Relational entities

The relational model is implemented with SQLAlchemy declarative models [18]. Each model represents a stable application concept: users, datasets, series, observations, model catalogue entries, experiments, runs, predic-

tions, and metrics.

The core run tables are implemented in `backend/app/models/experiment.py`. The `Experiment` model contains `id`, `user_id`, `name`, `series_id`, `model_id`, `horizon`, `use_mcp`, `config`, `status`, `created_at`, and `updated_at`. The `Run` model contains `id`, `experiment_id`, `started_at`, `finished_at`, `status`, `error_message`, and `created_at`. The output tables are also explicit: `Prediction` stores `id`, `run_id`, `timestamp`, `value`, `lower_ci`, and `upper_ci`; `Metric` stores `id`, `run_id`, `name`, `value`, and `created_at`.

Table 7.6.: Main PostgreSQL entities

Entity	**Main fields**	**Role in the platform**
User	email, password hash, role, status	Represents platform users and supports access-control decisions.
Dataset	slug, name, frequency, source path, version	Registers a dataset available to the platform.
Series	dataset, name, slug, unit	Represents a specific time series inside a dataset.
Observation	series, timestamp, value	Stores monthly numerical observations for a target or feature series.
Model catalogue	slug, name, model type, MCP support	Defines the models that can be selected by experiments.
Experiment	user, series, model, horizon, MCP flag, config, status	Stores the forecasting setup selected by the user.
Run	experiment, status, start/end timestamps, error	Represents one execution of an experiment.
Prediction	run, timestamp, value, confidence bounds	Stores the forecast values produced by a run.
Metric	run, name, value	Stores evaluation values such as MAE, RMSE, or MAPE for a run.

The relation between these entities preserves traceability. A metric belongs to a run, a run belongs to an experiment, an experiment points to a model and a series, and the series belongs to a dataset. This structure makes it possible to inspect a result and reconstruct the configuration that produced it.

7.7.2. Dataset and model catalogue

The dataset layer registers the data that can be used by the platform. A dataset contains one or more series, and each series contains timestamped observations. This structure is flexible enough to represent both target inflation series and auxiliary feature series. In the current platform, seeded datasets include the Spain CPI series, Global CPI, European HICP, and exogenous feature data.

The model catalogue separates model identity from model execution. The database stores the model slug, display name, type, MCP support flag, and active status. The actual execution logic lives in the forecasting adapter layer. The frontend can show available models from the database, while the backend resolves the selected slug to the appropriate adapter at execution time.

7.7.3. Experiment and run outputs

Experiments and runs are separated deliberately. An experiment is the definition of a forecasting setup. A run is one execution of that setup. This distinction allows the same experiment to be executed more than once, inspected over time, or compared with other experiments.

Predictions and metrics are stored as child entities of a run. This means that once a forecast has been executed, the frontend can retrieve the result without rerunning the model. The platform can also compare metric outputs across runs, display forecast curves, and attach contextual information to the execution record.

7.7.4. MongoDB context storage

MongoDB is used for news and contextual documents. In the run workflow, MCP context can be stored in a document associated with the run identifier. This allows the platform to preserve the contextual information retrieved for a forecast period, including macro signals and sentiment values when available.

The concrete write operation is in `backend/app/api/v1/runs.py`. When `use_mcp` is active and signals are returned, the backend performs a `replace_one` on the `mcp_contexts` collection using `{"run_id": run.id}` as the selector and `upsert=True`. The stored document has the following shape:

```
{
  "run_id": <generated_run_id>,
  "fetched_at": "<ISO-8601 timestamp>",
  "signals": [
    <signal_dictionaries_returned_by_the_MCP_client>
  ]
}
```

Pseudocode 7.3: MongoDB document shape for stored MCP context

This storage decision keeps contextual data flexible. News articles, MCP tool outputs, sentiment values, and availability flags do not always have the same schema as relational experiment entities. MongoDB avoids forcing semi-structured context into rigid relational tables, while still allowing the backend to connect the context to a run through a shared identifier.

The next section develops the forecasting adapter layer, which connects the persisted experiment configuration with the model execution logic.

7.8. FORECASTING ADAPTER LAYER

The forecasting adapter layer is the part of the backend that converts a stored experiment into an executable model call. It is deliberately separated from the API routers and from the database models. This keeps the forecasting code modular and makes it possible to add new models without rewriting the run lifecycle, the persistence logic, or the frontend workflow.

This layer is the technical bridge between model code and the software platform. It builds the architecture that can execute models, store their outputs, and expose them through a usable application.

7.8.1. Adapter contract

All forecasting adapters follow the same input and output contract. The input object contains the target time series, the forecast horizon, a configuration dictionary, and optional auxiliary structures. These optional structures include exogenous variables, previous run predictions for the stacking model, and stacking weights.

The exact dataclasses are defined in `backend/app/forecasting/base.py`. The `ForecastInput` object car-

ries the target series, the forecast horizon, and the configuration dictionary, together with the optional exogenous and stacking structures. The `ForecastResult` object carries the predictions, their timestamps, the train and test actuals, and the model slug. Table 7.7 lists these fields and the purpose of each one. The protocol requires every adapter to expose a `slug` attribute and a `run` method with the signature `run(inp: ForecastInput) -> ForecastResult`.

The output object contains the forecast values, their timestamps, the train and test actual values used by the execution, and the slug of the model that produced the result. With this uniform contract, the run executor does not need to know the internal details of ARIMA, SARIMAX, TimesFM, Chronos-2, TimeGPT, or the ensemble model. It builds the input, calls the selected adapter, and persists the returned predictions and metrics.

Table 7.7.: Forecast adapter contract

Element	**Main fields**	**Purpose**
`ForecastInput`	series, horizon, config	Defines the basic forecasting problem selected by the user.
`ForecastInput`	exog	Provides optional exogenous variables for models that can use contextual features.
`ForecastInput`	stack_preds, stack_weights	Provides previous forecast outputs and weights for the ensemble stacking adapter.
`ForecastResult`	predictions, timestamps	Returns the forecasted values and their associated monthly timestamps.
`ForecastResult`	train_actuals, test_actuals	Returns the observed values needed to calculate execution metrics.
`ForecastResult`	model_slug	Preserves the identity of the adapter that produced the result.

This design also simplifies error isolation. If a model fails, the failure remains inside its adapter or inside the run execution path. The API schema, the database model, and the frontend result page can remain stable as long as the adapter respects the same contract.

One concrete example is `NaiveSeasonalAdapter`, implemented in `backend/app/forecasting/adapters/naive.py` with slug `naive-seasonal`. The adapter sorts the input series, requires at least `13 + horizon`

observations, splits the last `horizon` observations as `y_test`, and predicts each future step from the value twelve months earlier in `y_train`. The run-test fixture in `tests/integration/conftest.py` inserts 48 synthetic monthly observations and uses this adapter with a 6-month horizon, so the adapter receives a 48-row monthly `pandas.Series`, returns 6 predictions, returns the 6 held-out timestamps, and sets `model_slug` to `naive-seasonal`.

7.8.2. Adapter registry

The backend uses a registry to map model slugs to adapter instances. The function that retrieves an adapter receives a slug and returns the corresponding implementation. If the slug is not registered, the run is marked as failed with an explicit error message.

The registry separates model metadata from executable model selection. The database catalogue stores name, type, description, and MCP support, while the registry stores the executable mapping. This avoids hard-coding model-specific branches throughout the API and keeps model selection concentrated in one place.

Table 7.8.: Forecasting adapters implemented in the backend

Slug	**Adapter**	**Role in the platform**
`naive-seasonal`	Naive seasonal	Simple seasonal baseline used as a reference point.
`arima`	ARIMA	Classical statistical model without explicit seasonality.
`auto-arima`	Auto ARIMA	Automatic ARIMA configuration for baseline comparison.
`sarima`	SARIMA	Seasonal statistical model for monthly inflation patterns.
`sarimax`	SARIMAX	Seasonal model with optional exogenous variables.
`ridge-exog`	Ridge with exogenous variables	Regression baseline for contextual feature usage.
`timesfm`	TimesFM	Foundation time-series model integrated through an adapter.
`chronos-2`	Chronos-2	Foundation time-series model integrated through an adapter.
`timegpt`	TimeGPT	External foundation forecasting model integrated through an adapter.
`ensemble-stack`	Ensemble stack	Inverse-MAE weighted combination of previous run predictions.

The registry gives the platform a clear extension point. To add another model, the backend needs a new adapter that follows the contract and a registry entry that maps its slug. This avoids embedding model-specific logic directly in the experiment or run endpoints.

7.8.3. Execution flow

When a run starts, the backend first retrieves the experiment and the selected model catalogue entry. It then changes the run status to `running`, loads the observations of the target series, and constructs a monthly `pandas` series. From that point, the executor prepares any optional inputs required by the selected model.

For `ridge-exog` and `sarimax`, the executor loads exogenous feature data from the database. For foundation models used with MCP enabled, it builds MCP-derived exogenous inputs from the contextual signals retrieved for the forecast horizon. For `ensemble-stack`, the executor loads predictions from previous runs and computes inverse-MAE weights, so that lower-error runs receive higher influence in the combined forecast.

Once the input object has been created, the adapter is executed in a background executor. This prevents a computationally expensive model call from blocking the asynchronous API event loop. After the adapter returns, the backend computes MAE, RMSE, and MAPE for the run, stores the forecast values as prediction rows, stores the metrics as metric rows, and marks the run as completed. If an error occurs, the run is marked as failed and the error message is preserved.

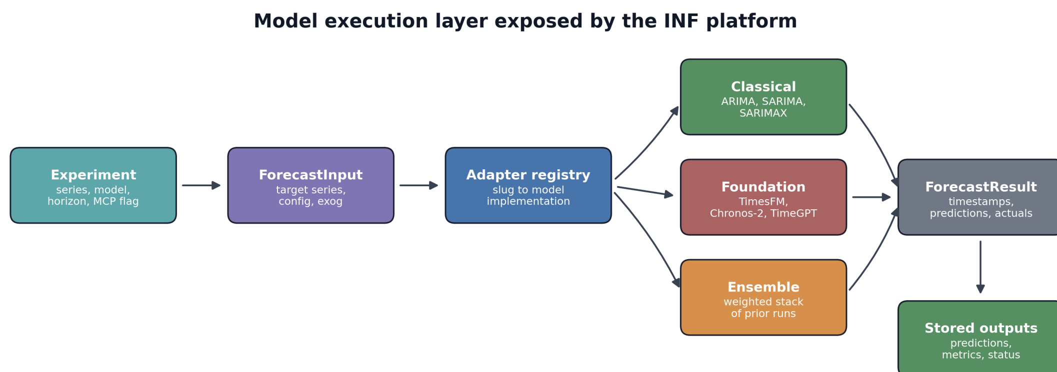


Figure 7.4.: Model execution layer exposed by the INF platform.

This execution design is intentionally practical. The platform-level metrics are sufficient for displaying and comparing run outputs in the web application. The INF contribution is the orchestration layer that makes those experiments reproducible from the application.

7.9. MCP INTEGRATION

The Model Context Protocol integration is treated as a software integration mechanism: a way to expose external context through a structured tool interface and connect it to the platform run lifecycle.

The platform does not assume that MCP signals always improve the forecast. It provides the infrastructure to retrieve them, store them, align them with forecast timestamps, and make them available to models and later inspection. Whether they improve performance is an empirical question handled by the forecasting study.

7.9.1. MCP server tools

The project includes a dedicated MCP server that runs through an SSE transport. The server exposes tools that the backend can call during a forecast run. Its role is to separate contextual data access from the main API backend, so that macro signals and news-derived context can be queried through a stable interface.

The tool definitions are implemented in `tfg-arquitectura/mcp_server/server.py`. The server registers three functions with `@mcp.tool()`: `get_macro_signals(year_month: str) -> str`, `get_news_context(start_date: str, end_date: str, topic: str = "inflacion") -> str`, and `get_news_sentiment(country: str, year_month: str) -> str`. The macro tool reads `tfg-forecasting/data/processed/mcp_signals_global.parquet`, which currently contains 276 monthly rows and nine columns: `fomc_hawkish_score`, `fomc_surprise_score`, `fomc_forward_guidance_num`, `ecb_hawkish_score`, `ecb_surprise_score`, `ecb_forward_guidance_num`, `us_cpi_surprise_score`, `us_cpi_direction_num`, and `us_cpi_components_pressure`.

Table 7.9.: MCP tools exposed by the platform

Tool	Input	Returned information
`get_macro_signals`	Month in `YYYY-MM` format	Pre-computed ECB, FOMC, CPI, and macro-context signals for the selected month.
`get_news_context`	Start date, end date, topic	Recent MongoDB news documents matching a topic and date range.
`get_news_sentiment`	Country and month	Monthly sentiment statistics and a hawkishness score derived from news articles.

The macro-signal tool reads the processed signal parquet file generated by the forecasting pipeline. The news-context tool searches the MongoDB collection containing raw news documents. The sentiment tool retrieves monthly news articles and applies a FinBERT sentiment pipeline, returning aggregate sentiment values rather

than the full article set.

```
{
  "year_month": "2024-12",
  "available": true,
  "signals": {
    "fomc_hawkish_score": 0.5,
    "fomc_surprise_score": 0.0,
    "fomc_forward_guidance_num": -1.0,
    "ecb_hawkish_score": 0.5,
    "ecb_surprise_score": 0.0,
    "ecb_forward_guidance_num": 0.0,
    "us_cpi_surprise_score": 0.0,
    "us_cpi_direction_num": 0.0,
    "us_cpi_components_pressure": 0.5
  }
}
```

Pseudocode 7.4: Example macro-signal record returned for 2024-12

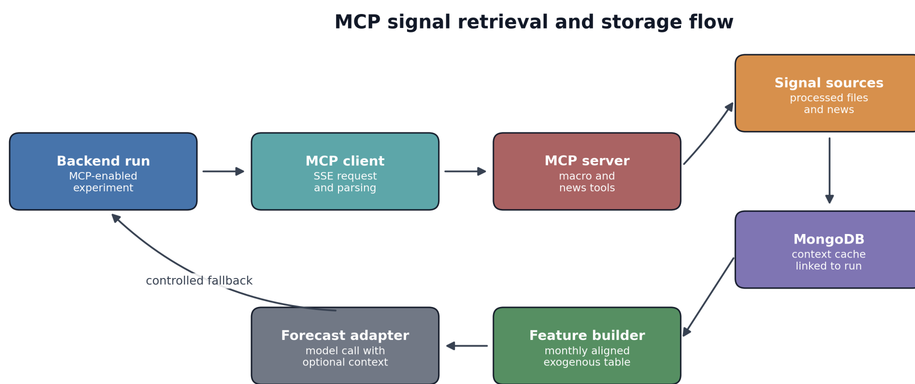


Figure 7.5.: MCP signal retrieval and storage flow in the INF platform.

Figure 7.5 represents how the backend contacts the MCP server, retrieves contextual signals, stores them in MongoDB, and connects them to a forecast run.

7.9.2. Backend MCP client

The backend contains an asynchronous MCP client used by the run executor. For each forecast timestamp, the client calls the macro-signal tool and the news-sentiment tool. The result is a list of monthly signal dictionaries that can be stored as run context and, when applicable, converted into exogenous inputs.

The client call shape is visible in `backend/app/mcp/client.py`. For each forecast timestamp, the backend calls `get_macro_signals` with `{"year_month": "YYYY-MM"}` and `get_news_sentiment` with `{"country": "spain", "year_month": "YYYY-MM"}` by default. It starts each row with `{"year_month": ym}`, merges available macro keys into that row, and adds `sentiment_mean`, `sentiment_std`, `sentiment_n`, and `sentiment_hawkish` when the sentiment tool returns data. The final persisted MongoDB document uses the `run_id`, `fetched_at`, and `signals` fields shown in Listing 7.3.

The client is designed with graceful degradation. If the MCP package is not available, the MCP server is unreachable, or a tool call fails, the client logs the problem and returns an empty result instead of cancelling the whole forecast run. Since MCP context is optional, a run should still complete when the selected model supports execution without it.

The client also normalizes the returned sentiment values. For each month, it can attach fields for the mean, standard deviation, article count, and hawkishness score. This gives the rest of the backend a simple tabular representation of news-derived context.

7.9.3. MCP-derived exogenous features

For models that can use contextual variables, the backend includes a feature builder that aligns MCP signals with the target time series. It loads the historical signal parquet, appends the future signals fetched through MCP when available, reindexes the result to the target series, forward-fills missing values, and removes columns with no useful variation.

This produces a dataframe that can be passed as an exogenous input to compatible adapters. In the case of foundation models such as TimesFM and Chronos-2, which do not expose the same native exogenous-variable interface as SARIMAX, the platform applies a damped Ridge-based residual correction. The correction estimates a signal-implied inflation level from historical MCP features and applies a limited adjustment to the base forecast.

The correction is intentionally conservative. Its purpose is not to hide the underlying model output or to claim

that contextual adjustment is always superior. It provides a controlled software mechanism for testing how MCP-derived signals could be attached to foundation-model forecasts inside the platform.

7.9.4. Context storage and traceability

When MCP signals are successfully retrieved, the backend stores them in the `mcp_contexts` MongoDB collection using the run identifier as the link to the relational execution record. The stored context remains inspectable after the run has finished. The frontend or later backend endpoints can retrieve it without calling the MCP server again.

This storage pattern improves traceability by connecting forecast values with the contextual signals available at execution time.

At the same time, the implementation avoids making MCP a single point of failure. Missing context, unavailable news articles, absent parquet files, or MCP connection errors are logged and handled without breaking the whole application. This supports experimentation with contextual forecasting while preserving repeated use.

7.10. FRONTEND APPLICATION

The frontend is the user-facing layer of the platform. Its purpose is to make the backend workflows understandable and usable through browser-based views. It is implemented as a React application built with Vite. React provides the component model used to compose pages and reusable interface elements, while Vite provides the development and build tooling for the web application [19, 20].

The frontend is part of the INF contribution because it defines how a user moves through the forecasting platform: selecting data, creating experiments, triggering runs, reading metrics, comparing models, inspecting MCP context, and exploring simulation views.

7.10.1. Application routing

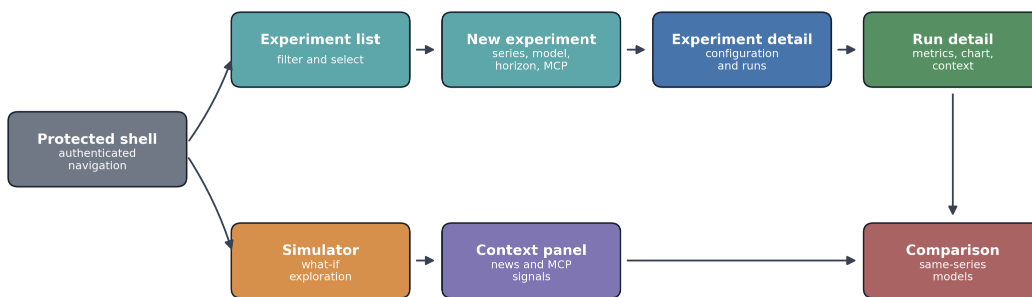
Client-side navigation is implemented with React Router, which provides the route structure for the single-page application [21]. The application separates public pages from protected platform pages. The landing, login, and signup pages are public. The experiment, run, comparison, simulator, and news-pulse pages are wrapped by a protected layout that requires an authentication token.

Table 7.10.: Main frontend routes

Route	Page	Purpose
/	Landing	Presents the platform entry point and links to the main workflows.
/login and /signup	Authentication pages	Allow users to access the protected part of the application.
/experiments	Experiments list	Shows existing experiments, grouped and filtered for inspection.
/experiments/new	New experiment	Provides the form for selecting dataset, series, model, horizon, and MCP usage.
/experiments/:id	Experiment detail	Shows experiment metadata, available runs, run status, and drift information.
/runs/:id	Run detail	Shows run status, predictions, metrics, MCP context, and generated narrative analysis.
/compare	Model comparison	Compares selected experiments through forecast overlays, metrics, skill score, and significance views.
/simulator	What-if simulator	Lets the user perturb contextual signals and inspect their effect on a counterfactual forecast.
/today	Inflation pulse	Shows recent inflation-related news and sentiment context.

This route structure mirrors the domain workflow of the platform. The user starts from datasets and experiments, moves into runs, and then uses comparison, context, and simulation views to interpret the results. This keeps navigation close to the concepts already stored in the backend.

Frontend navigation and main user workflows



The interface exposes backend concepts as user tasks rather than raw scripts or database records.

Figure 7.6.: Frontend navigation and main user workflows.

Figure 7.6 shows the main navigation paths between experiment creation, run execution, run inspection, comparison, simulation, and news-context views.

7.10.2. API access and state management

The frontend communicates with the backend through a small API wrapper that targets the `/api/v1` prefix. The wrapper centralizes JSON requests, authentication headers, error handling, and unauthorized-session behaviour. If the backend returns an unauthorized response, the stored token is removed and the user is redirected to the login page.

Server-state management is implemented with TanStack Query, which supports fetching, caching, refetching, and updating asynchronous server state in frontend applications [22]. In this project, it organizes data access around domain hooks for the main platform entities, MCP context, drift results, news sentiment, and what-if setup data.

This layer improves maintainability because pages do not build raw request logic by themselves. Instead, they call hooks such as `useExperiments`, `useRun`, `usePredictions`, `useRunMetrics`, or `useMcpContext`. These hooks keep the frontend close to the backend API while still giving each page a clear and readable structure.

7.10.3. Experiment creation workflow

The new-experiment page implements the first complete user workflow of the platform. The user selects a dataset, then a series from that dataset, then a model from the active model catalogue. The form also includes the forecast horizon and a switch for MCP usage. This mirrors the backend experiment entity and avoids asking the user to manually write JSON configuration for the common case.

The page groups models into statistical baselines and foundation models. Classical models such as ARIMA, SARIMA, SARIMAX, and Ridge are displayed together, while TimesFM, Chronos-2, and TimeGPT are shown as foundation models.

The page also includes a JSON payload preview. The preview makes the relation between the user interface and the backend API explicit. The user sees that the form will become a request to create an experiment with a name, series identifier, model identifier, horizon, and MCP flag.

7.10.4. Run inspection

The run-detail page is the main inspection surface for a completed or running forecast. It retrieves the run status, prediction values, metrics, MCP context, and optional narrative analysis through the endpoint described in Subsection 7.6.5. If a run is still pending or running, the page refreshes periodically so the user can follow execution progress without manually reloading the browser.

The page shows MAE, RMSE, and MAPE as metric cards, followed by a forecast trajectory chart. The charting components are built with Recharts, a React charting library based on reusable chart components [23]. In this platform, Recharts is used to show forecast curves, sentiment timelines, what-if forecasts, comparison overlays, metric bars, radar summaries, and other result-oriented visualizations.

When MCP context is available, the run-detail page displays the number of signal rows retrieved, a sentiment timeline if sentiment values exist, and a compact preview of the stored context. This connects the visual result with the contextual retrieval described in the previous section.

7.10.5. Model comparison dashboard

The comparison page provides a broader view across experiments. It restricts comparison to experiments from the same target series, which avoids mixing forecasts that are not directly comparable. The user selects experiments from a sidebar, and the page builds the comparison dataset by combining experiment metadata, run details, predictions, metrics, actual observations, and benchmark values.

The dashboard includes several views: forecast overlay, horizon-wise error behaviour, skill score against the seasonal naive benchmark, leaderboard, Diebold-Mariano significance matrix, metric bars, radar summary, and accuracy-runtime scatter. These views give users an interactive way to inspect comparable forecasting evidence.

The implementation also limits the number of selected experiments in the interface. This is a small usability decision but it matters for readability. Model-comparison charts can become confusing if too many runs are shown at once, so the interface nudges the user toward focused comparisons.

7.10.6. Simulation and context views

The simulator page exposes a what-if workflow based on contextual signals. The user selects a dataset, a series, and a forecast horizon, then adjusts signal sliders to inspect how the counterfactual forecast changes. The page separates the baseline forecast from the counterfactual forecast and shows summary cards for short-horizon and final-horizon changes.

The simulator is an exploratory interface, not a causal economic model. Its role in the INF thesis is to demonstrate that the platform can expose contextual variables through a controlled user interface and connect them with forecast visualizations.

The inflation-pulse page complements this workflow by showing recent inflation-related news and sentiment context. Together with the run-detail MCP panel, it gives the platform a visible connection between numerical forecasts and the textual or macroeconomic context retrieved through the MCP service.

7.10.7. Frontend design decisions

The frontend design follows a dashboard-oriented structure. The main screens emphasize scanning, comparison, and repeated use through lists, filters, tables, metric cards, charts, status indicators, and compact contextual panels.

The interface also uses typed domain objects through frontend type definitions. These types give the frontend a shared vocabulary for platform entities, MCP context, and what-if points exchanged with the backend.

Overall, the frontend completes the platform loop by turning datasets, models, runs, metrics, context, and comparisons into workflows that can be used and evaluated.

7.11. EXECUTION ENVIRONMENT AND REPRODUCIBILITY

The platform is composed of several services that must work together: backend, frontend, gateway, relational database, document database, MCP server, and experiment-tracking server. To make this environment reproducible, the repository includes a Docker Compose configuration that defines the main services, ports, environment variables, dependencies, and persistent volumes.

Docker supports this project by reducing differences between local machines and making service dependencies

explicit [11, 24]. Docker Compose extends this idea by defining and running multi-container applications from a single configuration file [12]. In this project, Compose provides a reproducible development and demonstration environment for the implemented architecture.

7.11.1. Compose services

The root `docker-compose.yml` file defines the platform as a coordinated service stack. Table 7.11 summarizes the main services included in the current environment.

Table 7.11.: Services defined in the Docker Compose environment

Service	Main image or build	Responsibility
`postgres`	`postgres:16-alpine`	Stores structured platform entities in the relational database.
`mongo`	`mongo:7`	Stores or serves news and contextual documents used by the MCP and news workflows.
`mlflow`	MLflow image	Provides the experiment-tracking service used to log run parameters and metrics.
`backend`	Backend Dockerfile	Runs the FastAPI application, forecasting adapters, database access, MCP client, and API routes.
`frontend`	Frontend Dockerfile	Runs the Vite development server for the React user interface during local execution.
`gateway`	Gateway Dockerfile	Builds the frontend and serves it through Nginx, while proxying API requests to the backend.
`mcp_server`	MCP server Dockerfile	Runs the MCP server that exposes macro signals, news context, and news sentiment tools.

The same Compose file exposes concrete ports and internal service URLs. PostgreSQL maps `${POSTGRES_PORT:-5432}:5432`; MongoDB maps `27017:27017`; MLflow maps `5000:5000`; the backend maps `${BACKEND_PORT:-8000}:8000`; the frontend maps `3000:3000`; the gateway maps `80:80`; and the MCP server maps `${MCP_SERVER_PORT:-8080}:8080`. The backend service receives `MCP_SERVER_URL=http://mcp_server:8080/sse`, `MONGO_URI=mongodb://mongo:27017/tfg_news`, `MLFLOW_TRACKING_URI=http://mlflow:5000`, and `PYTHONPATH=/app/tfg-arquitectura/backend:/app`. It also mounts `./tfg-arquitectura/backend`, `./tfg-forecasting`, and `./shared` into the container. The MCP server mounts `./tfg-forecasting/data/processed` read-only and keeps a Hugging Face cache volume for FinBERT/model downloads.

This service composition reflects the architecture described earlier in the chapter. The backend is not responsible for hosting every dependency internally. Instead, each major concern is placed in its own service: relational persistence, document persistence, experiment tracking, API execution, frontend execution, reverse proxy, and contextual tool serving.

7.11.2. Configuration management

The platform uses environment variables for configuration. The repository includes an `.env.example` file with the main values required to run the stack: PostgreSQL credentials, MongoDB connection string, JWT settings, external API keys, MCP server port, backend port, and administrator seed credentials.

The example configuration defines `POSTGRES_USER=tfg`, `POSTGRES_PASSWORD=changeme`, `POSTGRES_DB=tfg_experiments`, `POSTGRES_HOST=postgres`, `POSTGRES_PORT=5432`, `MCP_SERVER_PORT=8080`, `JWT_ALG=HS256`, `JWT_EXPIRES_MIN=10080`, `ADMIN_EMAIL=admin@tfg.local`, and `BACKEND_PORT=8000`. Empty keys such as `NIXTLA_API_KEY` and `GDELT_API_KEY` are left for the local operator to fill when external services are needed.

The backend reads these values through a settings object based on environment configuration. It builds the PostgreSQL connection URL from the individual database fields, reads the MongoDB URI, receives the MCP server URL, and uses the MLflow tracking URI. This keeps deployment-specific values out of the application code and supports local, containerized, or later deployment environments.

Some configuration values are intentionally development-oriented. For example, the example JWT secret and administrator password are placeholders, and the frontend development service uses bind mounts and a Vite development server. Before deployment outside the local environment, secrets, network exposure, CORS policy, and service hardening would need to be reviewed.

7.11.3. Containers and mounted code

The backend image is built from a Python 3.11 base image with `uv` [25]. It installs the FastAPI stack, database libraries, forecasting dependencies, MCP client support, MLflow, and the model-related libraries required by the adapters. Heavy machine-learning dependencies such as CPU PyTorch, Chronos, and TimesFM are installed in a separate layer, which makes the image structure more maintainable when base dependencies change.

During local Compose execution, the backend mounts the backend source code, the forecasting repository, and the shared utilities into the container. The service can access both the platform code and the forecasting artifacts without duplicating them. The `PYTHONPATH` also includes the backend and repository root so that backend modules and shared utilities can be imported consistently.

The MCP server follows a similar pattern. Its image installs MCP support, data-processing libraries, MongoDB access, and the transformer stack needed for FinBERT sentiment scoring. It mounts the processed forecasting

data directory as read-only and keeps a Hugging Face cache volume so model downloads do not need to be repeated unnecessarily.

7.11.4. Gateway and network boundary

The gateway service builds the React application and serves the generated static files through Nginx [26]. It also proxies requests under `/api/` to the backend service. This creates a cleaner external entry point: the browser can interact with one web server, while the gateway routes API traffic internally to FastAPI.

The Nginx configuration also includes a single-page-application fallback. Requests that do not match a static asset are routed back to `index.html`, allowing frontend routes such as `/experiments`, `/runs/:id`, or `/compare` to work when the browser is refreshed directly on those paths.

Inside the Compose network, services refer to each other by service name. For example, the backend uses `mongo`, `postgres`, `mlflow`, and `mcp_server` as internal hostnames. This keeps the application configuration close to the service architecture and avoids hard-coding local host addresses inside the backend code.

7.11.5. Persistent volumes

The Compose environment defines persistent volumes for PostgreSQL, MongoDB, MLflow, and the Hugging Face cache. These volumes matter because the platform produces state during execution. Platform entities, contextual documents, MLflow run metadata, and downloaded model files should not disappear every time a container is recreated.

At the same time, this persistence is scoped to the local Compose environment. A future deployment would need backups, migration policies, monitoring, access control, and explicit data-retention decisions.

7.11.6. Experiment tracking with MLflow

The platform includes an MLflow service as a first experiment-tracking layer. MLflow Tracking is designed to log parameters, metrics, artifacts, and related information from machine-learning runs so they can be inspected later [10, 27]. In this implementation, the backend sets the tracking URI, selects a dedicated platform experiment, and logs run parameters and metrics after a forecasting adapter finishes.

The logged parameters include the model slug, forecast horizon, MCP flag, series identifier, experiment iden-

tifier, run identifier, and number of MCP signal rows. The logged metrics correspond to the platform-level run metrics computed by the backend. MLflow complements the relational database: PostgreSQL stores the application entities and user-facing outputs, while MLflow provides a tracking-oriented view of model executions.

The exact logging block in `backend/app/api/v1/runs.py` sets the tracking URI from `settings.MLFLOW_TRACKING_URI`, selects the experiment `tfg-forecasting-platform`, and starts runs named `run-{run.id}-{model_cat.slug}`. It logs seven parameters: `model_slug`, `horizon`, `use_mcp`, `series_id`, `experiment_id`, `run_id`, and `n_mcp_signals`. It logs the metrics dictionary computed by the backend, whose current keys are `mae`, `rmse`, and `mape`, and it sets the tag `platform.run_id`.

```
{
  "run_name": "run-<run_id>-<model_slug>",
  "experiment": "tfg-forecasting-platform",
  "params": {
    "model_slug": "<model_catalogue_slug>",
    "horizon": <experiment_horizon>,
    "use_mcp": <experiment_mcp_flag>,
    "series_id": <series_id>,
    "experiment_id": <experiment_id>,
    "run_id": <run_id>,
    "n_mcp_signals": <number_of_retrieved_MCP_rows>
  },
  "metrics": {
    "mae": <computed_MAE>,
    "rmse": <computed_RMSE>,
    "mape": <computed_MAPE>
  },
  "tags": {
    "platform.run_id": <run_id>
  }
}
```

Pseudocode 7.5: MLflow parameters and metrics logged for one run

MLflow errors are handled without failing the whole run. If tracking is unavailable, the backend logs a warning and continues storing predictions and metrics in PostgreSQL. Optional observability services improve traceability without becoming a single point of failure for forecast execution.

7.11.7. Reproducibility limits

The containerized environment improves reproducibility, but it does not make every result perfectly deterministic. Forecasting models can depend on external services, model-package versions, API keys, downloaded weights, and the contents of the processed data directory. TimeGPT requires an external API key. Foundation-model packages and FinBERT weights may change over time unless pinned and cached carefully. News data can also vary depending on when it is retrieved.

The platform combines several reproducibility mechanisms. The repository organizes the code, Compose defines the service topology, environment variables record configuration, PostgreSQL stores run outputs, MongoDB stores contextual documents, and MLflow logs run parameters and metrics. Together, these mechanisms make the system inspectable and repeatable.

7.12. VALIDATION AND TESTING

Validation in the INF thesis focuses on the software system: API behaviour, permissions, persistence workflows, forecast-run execution, graceful failure, and operational checks. The objective is to verify that the platform behaves consistently as an application and that the main user workflows are supported by tests or explicit checks.

The backend test suite is implemented with pytest and asynchronous fixtures [28]. pytest fixtures provide reusable setup and teardown logic [29], which is useful here because many tests need a clean database, authenticated users, seeded datasets, model catalogue entries, and synthetic time-series observations. FastAPI applications can be tested through clients that exercise the application without starting a separate HTTP server [30]; this project uses an asynchronous HTTP client with an ASGI transport to call the application directly.

The integration suite contains 84 collected tests under `tfg-arquitectura/backend/tests/integration`. The file-level breakdown is: 8 authentication-flow tests in `test_auth_flow.py`, 13 dataset tests in `test_datasets.py`, 21 experiment tests in `test_experiments.py`, 3 health tests in `test_health.py`, 11 metric/comparison tests in `test_metrics.py`, 21 run-lifecycle tests in `test_runs.py`, and 7 user/role tests in `test_users.py`.

7.12.1. Test organization

The backend tests are located under `tfg-arquitectura/backend/tests`. The shared test configuration creates an asynchronous API client and resets the relevant PostgreSQL tables before each test. The integration

fixtures create users with different roles, seeded datasets, series, model catalogue rows, and synthetic monthly observations.

At the time of writing, the integration suite contains 84 test functions. This number is less important than the type of coverage it represents. The tests focus on the workflows that would most easily break the platform if the API, database model, permissions, or run lifecycle changed unexpectedly.

Table 7.12.: Main backend validation areas

Area	**Representative tests**	**Risk addressed**
Authentication	signup, duplicate signup, login, token-based `/me`	Confirms that user access and token flows work as expected.
Users and roles	promote/demote users, forbidden role changes, admin self-demotion	Checks that role-based permissions cannot be bypassed through the API.
Datasets and catalogues	list datasets, series, observations, models, pagination, not-found cases	Verifies that controlled data sources can be inspected safely.
Experiments	create, list, detail, delete, ownership, invalid series/model, invalid horizon	Validates the main experiment-management workflow.
Runs	trigger runs, pending status, completed runs, failed runs, predictions, metrics	Tests the execution lifecycle from experiment to stored outputs.
Metrics comparison	compare experiments, ordering, deduplication, authorization, response shape	Supports the model-comparison views consumed by the frontend.
Robust failure paths	unsupported model, missing foundation-model dependencies, missing API key, unavailable MCP	Confirms that external or optional capabilities fail visibly instead of corrupting the platform state.
Health checks	status response, timestamp, database status	Provides a basic operational check for the running service.

This coverage aligns with the requirements defined earlier in the chapter. It validates the behaviours that matter for the platform as a user-facing system: authentication, experiment creation, run execution, output retrieval, metric comparison, and authorization.

7.12.2. Unit-level checks

The integration suite is complemented by smaller unit tests that exercise pure or low-dependency logic. Under `tfg-arquitectura/backend/tests/unit`, the tests validate shared metric functions, the forecasting adapter registry, and the residual-drift computation used by the drift endpoint. These checks are useful because they isolate behaviours that should remain stable even when the database, API fixtures, or frontend are not involved.

The forecasting area also contains lightweight metric tests under `tfg-forecasting/tests/test_metrics.py`. They check MAE, RMSE, MASE, Diebold-Mariano comparison behaviour, and the consistency of the summary helper against individual metric functions. This gives the project a deterministic test layer for the evaluation vocabulary shared by the CDIA pipeline and the INF platform.

7.12.3. Database and fixture isolation

The test suite uses database isolation by truncating the main relational tables before each test. This includes users, datasets, series, observations, model catalogue rows, experiments, runs, predictions, and metrics. Resetting identity values reduces coupling between tests.

The fixtures create controlled data rather than relying on the full real forecasting datasets. For run-related tests, a synthetic monthly series with 48 observations is inserted. This allows adapters such as the seasonal naive model and SARIMA to be exercised quickly, while keeping the tests small enough for repeated development use.

This strategy is appropriate for platform validation. The tests do not reproduce the full forecasting study; they verify that the API and persistence layers behave correctly with a realistic enough time series and model catalogue entry.

7.12.4. Run lifecycle validation

The run tests validate several important behaviours. A triggered run returns a `202 Accepted` response and starts in a pending state. Simple adapters can complete and produce stored predictions and metrics. Unsup-

ported model slugs lead to a failed run with an explicit error message. A series without observations returns a validation error instead of starting an impossible execution.

The tests also verify access control around runs. Requests without authentication fail. Requests made by a different non-admin user are forbidden. These cases matter because runs are not isolated technical records; they are linked to experiments and users.

The suite includes graceful-failure checks for optional model families. TimesFM, Chronos-2, and TimeGPT execution paths are tested so that missing libraries or missing external API keys produce failed run states rather than silent errors. The MCP-unavailable path is also tested: enabling MCP must not prevent a run from completing when the selected model can run without retrieved context.

7.12.5. Metric and comparison validation

The metrics tests validate the comparison endpoint used by the frontend dashboard. They check that completed runs return MAE and RMSE values, that experiments without completed runs are represented explicitly, that the order of requested experiments is preserved, and that duplicated experiment identifiers are deduplicated.

Authorization is also tested in the comparison workflow. A researcher cannot compare another user's experiment, while an administrator can inspect experiments across users. The endpoint also rejects too many experiment identifiers. The comparison dashboard depends on this endpoint to build its visual summaries.

7.12.6. Operational checks and drift

The backend includes a health endpoint that returns the service status, a timestamp, and the database status. If the database check fails, the response becomes degraded. This gives the Compose environment and the developer a simple way to verify whether the backend can reach PostgreSQL.

The platform also includes a first residual-drift endpoint. It takes the most recent completed run of an experiment, computes residuals where actual observations are available, splits them into early and recent windows, and applies a two-sample Kolmogorov-Smirnov test. Drift detection is relevant because deployed machine-learning systems can degrade when the data observed during use changes over time [14]. In this project, the drift endpoint is an initial monitoring mechanism.

The endpoint is implemented as `GET /api/v1/drift?experiment_id={id}` in `backend/app/api/v1/`

drift.py. It returns experiment_id, run_id, drifted, p_value, ks_statistic, n_early, n_recent, and message. The code uses the latest RunStatus.done run for the experiment, computes residuals as prediction minus actual observation for matching timestamps, uses the first 60% of residuals as the early window and the last 40% as the recent window, and applies scipy.stats.ks_2samp with alpha 0.05. A real Compose-backed call to GET /api/v1/drift?experiment_id=12 returned experiment_id=12, run_id=11, drifted=true, p_value=0.0025, ks_statistic=1.0, n_early=7, n_recent=5, and the message Drift detected (KS=1.000, p=0.0025 < 0.05). A contrast call for experiment_id=3 returned drifted=false, p_value=0.5455, and ks_statistic=0.4286.

7.12.7. Continuous integration pipeline

The repository already includes a GitHub Actions workflow in .github/workflows/ci.yml. The workflow runs on pushes to main and refactor-clean, and also on pull requests. It is divided into three jobs that check different parts of the project.

The first job, backend-lint, installs Ruff in a Python 3.11 environment and runs ruff check app scripts tests inside tfg-arquitectura/backend. This validates the backend source, backend scripts, and backend tests at linting level. The second job, forecasting, installs Ruff, pytest, NumPy, and SciPy, then runs ruff check shared tfg-forecasting/tests and pytest tfg-forecasting/tests/test_metrics.py -q. This gives the forecasting side a lightweight automated check for shared code style and deterministic metric behaviour. The third job, frontend, uses Node 20, installs frontend dependencies with npm ci, runs npm run lint, and builds the React/TypeScript application with npm run build.

This CI pipeline does not replace the full Compose-backed validation described above. Its current role is to catch style, deterministic metric, and frontend build regressions quickly, while the backend integration suite and real drift endpoint checks remain heavier validation evidence executed in the local service environment.

7.12.8. Validation limits

The implemented validation is incomplete. The backend integration tests cover the main API behaviours, permissions, and run workflows, and the existing CI pipeline checks backend linting, deterministic forecasting metric tests, frontend linting, and frontend build success. However, the project does not yet include a full browser-based end-to-end suite for the React frontend. The visual dashboards, chart interactions, and simulator controls are validated mainly through implementation inspection and manual use, not through automated UI tests.

The current tests also do not perform load testing, security penetration testing, or long-running reliability tests. External services such as TimeGPT, live news retrieval, model downloads, and full MCP sentiment execution are difficult to make deterministic in a lightweight integration suite. For that reason, the platform favours graceful failure and explicit status reporting for these paths.

These limits define the current validation boundary. The platform demonstrates tested backend workflows, reproducible service orchestration, traceable persistence, and observable run states. A future version should extend this with frontend end-to-end tests, load tests, stronger security review, and automated deployment checks.

7.13. CONNECTION WITH THE FORECASTING PIPELINE

The INF platform connects with the forecasting work and makes it executable, persistent, and inspectable through software architecture. For this reason, the development includes a deliberate integration boundary between `tfg-forecasting`, `tfg-arquitectura`, and `shared`.

The boundary is implemented at repository level. The `tfg-forecasting` area preserves the analytical scripts and artifacts, `tfg-arquitectura` contains the backend, frontend, gateway, MCP server, and deployment code, and `shared` contains common utilities used across the project. This separation lets the platform expose forecasting artifacts without absorbing the whole research pipeline into the web application.

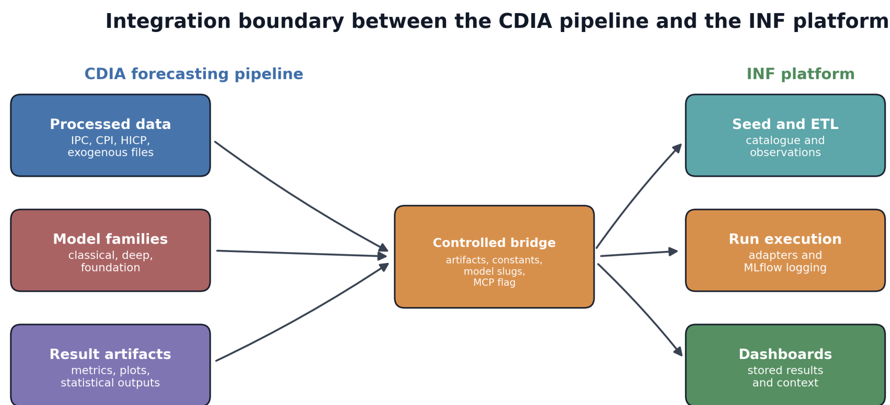


Figure 7.7.: Integration boundary between the forecasting pipeline and the INF platform.

Figure 7.7 shows the forecasting pipeline on one side and the platform on the other. It includes processed datasets, result artifacts, shared metrics and constants, backend seed and ETL scripts, PostgreSQL, MLflow, MCP signals, and frontend dashboards, with emphasis on the implemented data and service connections.

7.13.1. Integration principle

The integration follows a simple principle: the forecasting project remains the analytical source, and the INF platform becomes the operational interface. The platform does not rewrite the complete experimental pipeline inside the backend. Instead, it takes the outputs and contracts that are useful for an application: processed time-series files, series identifiers, model slugs, forecast horizons, metric definitions, contextual signal files, and result summaries.

This decision separates two responsibilities. The forecasting pipeline explores alternatives, runs evaluations, compares conditions, and studies model behaviour. The software platform exposes stable workflows, persists state, applies permissions, provides user interaction, and makes outputs inspectable.

The separation still preserves a practical bridge between both TFGs. A user can create an experiment, choose a model, trigger a run, inspect predictions, compare metrics, and view contextual information without manually navigating the forecasting folders.

7.13.2. Processed data artifacts

The first connection point is the processed data layer. The forecasting pipeline produces curated Parquet files in the processed data directory. These files are the stable hand-off between the analytical pipeline and the platform. They include the Spanish IPC index, global CPI series, European HICP series, exogenous macroeconomic features, and MCP-related signal files.

The backend includes explicit seed and ETL scripts that load these files into the relational database. The script `backend/seed_ipc.py` defines the main application datasets and maps each Parquet file to dataset metadata, series slugs, human-readable names, units, and timestamps. It loads the data with `pandas.read_parquet`, normalizes the time index, creates or refreshes series, and inserts observations into PostgreSQL. The lower-level ETL module `backend/app/etl/load_parquets.py` follows the same idea for selected processed datasets and model catalogue entries. pandas provides direct support for loading Parquet files into data frames, which fits this kind of structured time-series ingestion workflow [31].

Table 7.13.: Main forecasting artifacts consumed by the INF platform

Artifact	**Repository location**	**Role in the platform**
Processed target series	`tfg-forecasting/data/processed`	Seeded into PostgreSQL as datasets, series, and observations.
Exogenous features	`features_exog.parquet`	Used by adapters and platform experiments that require macroeconomic explanatory variables.
MCP signal files	Processed MCP signal Parquets	Provide contextual signals used by MCP-enabled workflows.
Model result files	`tfg-forecasting/08_results`	Preserve forecasting evaluation outputs, prediction files, metric summaries, plots, and comparison artifacts.
Shared constants	`shared/constants.py`	Keep common series identifiers, date windows, conditions, frequency, and forecast horizon aligned.
Shared metrics	`shared/metrics.py`	Provide common metric definitions such as MAE, RMSE, MASE, and Diebold-Mariano support.

This artifact-based connection avoids manually copying values into the backend. The database receives data from generated files, while the original analytical artifacts remain available for inspection in the forecasting folder. If a dataset or result appears in the platform, its source can be related back to the forecasting pipeline that produced it.

7.13.3. Dataset and catalogue seeding

The platform uses seeding as the practical mechanism that turns forecasting artifacts into application state. The seed scripts do more than insert rows. They define the names by which the user interface and API understand

the forecasting domain.

For example, `seed_ipc.py` maps the Spanish IPC processed file to the application dataset `ipc-spain-ine`. Inside that dataset, `indice_general` becomes the main target series and the IPC categories are registered as additional series. The same script also registers global CPI, European HICP, and exogenous-feature datasets. This gives the frontend a clean catalogue of datasets and series, instead of exposing raw file names directly to the user.

The model catalogue follows the same pattern. Catalogue rows expose stable model slugs for baseline, statistical, exogenous, foundation-model, and ensemble executions. These slugs are the contract between the frontend, backend, adapter registry, and experiment records, without requiring the user to know the internal script organization of the forecasting repository.

This seeding approach also supports repeated local execution. Depending on the script, loaders either skip existing records or refresh observations. Developers can rebuild the platform database during development without manually recreating the catalogue. It also fits the Compose environment, where databases may be recreated while the source artifacts remain mounted from the repository.

7.13.4. Experiment conditions and MCP connection

The forecasting workflow distinguishes between numerical-history-only executions and executions enriched with contextual or MCP-derived information. The INF platform reflects this distinction through the experiment field `use_mcp`, the adapter inputs, and the contextual panels shown in the frontend.

The script `seed_experiments.py` creates experiments for selected models with and without MCP. The script `seed_ablation.py` focuses on the foundation-model comparison between no-MCP and MCP configurations. These scripts show how experimental conditions can be instantiated as application experiments and runs.

When MCP is enabled, the backend retrieves contextual signals through the MCP client and builds exogenous context for compatible adapters. If the MCP service is unavailable, the run lifecycle fails gracefully or continues when the selected model can execute without context.

The MCP server also consumes processed forecasting data through a read-only mount defined in the Compose configuration. The contextual service can access processed signal files without mutating the forecasting source directory. The platform keeps a clear data direction: forecasting artifacts feed the architecture services, while the application stores its own runs, predictions, metrics, and contextual records separately.

7.13.5. Result artifacts and platform outputs

The forecasting repository contains a large `08_results` directory with metrics, prediction files, summaries, statistical-comparison outputs, and figures. These artifacts guide which model families, metrics, conditions, and visualizations the INF application needs to support.

The platform does not import every historical result file as a first-class database object. Instead, it implements the structures needed to generate and store platform runs: experiments, runs, predictions, and metrics. This is a deliberate development decision. Importing all prior result files would have increased the data-migration burden and could have blurred the difference between an offline research result and an application execution.

The current design has two complementary result layers. The analytical layer preserves the result archive under `08_results`. The platform layer stores application-triggered run outputs in PostgreSQL and logs selected parameters and metrics to MLflow. Together, both layers preserve traceability without forcing all result artifacts into a single storage model.

7.13.6. Shared utilities

The `shared` folder is a small but important part of the integration. It contains constants that define common series identifiers, date windows, forecast horizon, monthly frequency, and experimental conditions. It also contains metric functions for MAE, RMSE, MASE, and Diebold-Mariano comparison support.

This shared layer reduces semantic drift between the forecasting pipeline and the platform. If the two areas use different names for the same series or different interpretations of the same horizon, the project becomes harder to validate. Shared constants and metric definitions help keep the terminology and evaluation vocabulary aligned.

The shared layer is intentionally limited. It does not become a large framework, and it does not hide the main logic of either project area. Its role is to keep the most reusable concepts in one place while allowing the forecasting and platform areas to maintain their own responsibilities.

7.13.7. Traceability across platform layers

The connection with the forecasting pipeline gives the INF platform its main motivation: datasets, models, runs, metrics, and contextual signals become a usable software system.

This traceability is visible in several development decisions. The dataset catalogue corresponds to processed forecasting files. The model catalogue corresponds to the model families used by the platform. The `use_mcp` flag corresponds to contextual-condition logic. The comparison dashboard corresponds to the need to inspect metrics across models and runs. The MLflow integration corresponds to the need for reproducible experiment tracking. The drift endpoint corresponds to the idea that forecast systems must be monitored after execution, even if the current implementation remains a first approximation.

The final result is an engineering continuation of the forecasting study. The INF development turns the forecasting pipeline into a system with APIs, persistence, execution workflows, contextual services, dashboards, and reproducible infrastructure.

7.14. MAINTAINABILITY AND EXTENSION POINTS

The last development concern is maintainability. The platform needed a structure that could be understood, extended, and validated after the initial implementation. This is especially relevant in forecasting systems, where model code is only one part of the total system. Prior work on ML technical debt highlights risks linked to data dependencies, configuration, weak boundaries, monitoring gaps, and undeclared consumers [7]. The INF development addresses this by making the main responsibilities explicit.

7.14.1. Module boundaries

The repository organization creates the first maintainability boundary. The forecasting study remains under `tfg-forecasting`, the platform implementation remains under `tfg-arquitectura`, and common definitions are placed under `shared`. This allows the analytical pipeline to change without directly changing the frontend, and the platform to improve without rewriting the experiment scripts.

Within that platform area, the frontend is implemented as a React and TypeScript application under `tfg-arquitectura/frontend`. This location contains the Vite application, page components, reusable UI components, API-query helpers, styling files, and frontend build configuration.

Inside the backend, maintainability is supported by domain routers, separated database models, schemas for request and response shapes, database dependencies, and forecasting adapters behind a common interface. The adapter registry prevents the run-execution endpoint from containing model-specific logic for every possible forecasting family. A new model still requires careful implementation, but it has a clear place in the codebase.

The frontend follows a similar pattern. Pages represent user workflows, while API functions and state hooks isolate communication with the backend. A new comparison view, contextual panel, or simulation page can reuse the existing API access layer instead of building request logic from scratch.

7.14.2. Security and access-control considerations

The platform includes a basic security model. Passwords are stored as bcrypt hashes, authentication uses signed JWT tokens, and protected endpoints retrieve the current user through backend dependencies. Role checks distinguish `admin`, `researcher`, and `viewer` workflows. Experiments and runs also apply ownership checks, so a non-admin user cannot inspect or manipulate another user's experiment records.

This is relevant because API-based applications often expose object identifiers in routes, and object-level authorization must be checked server-side for each protected object. OWASP identifies broken object-level authorization as a major API security risk [32]. In this platform, that risk is addressed at the development level by checking ownership in experiment and run endpoints before returning data such as predictions, metrics, or stored MCP context.

Security hardening remains future work. Before deployment outside a controlled environment, the default values in `.env.example`, permissive CORS settings, local admin credentials, and development tokens would need to be reviewed.

7.14.3. Operational robustness

Several parts of the implementation were designed to degrade visibly when an optional service is unavailable. If MCP retrieval fails during a run, the backend records a warning and continues when the selected adapter can run without contextual data. If MLflow tracking fails, the run can still store predictions and metrics in PostgreSQL. If Ollama is unavailable for narrative generation, the narration endpoint returns a service-unavailable error instead of corrupting the run state.

The platform depends on services with different reliability profiles: PostgreSQL, MongoDB, MLflow, MCP server, local or external model dependencies, and optional language-model narration. The implementation separates core run persistence from optional observability or explanation services.

The run status model also improves robustness. A run is not treated as a simple synchronous request. It has a lifecycle with pending, running, done, and failed states. This gives the frontend and the user an explicit repre-

sentation of long-running or failed executions, and it keeps the database as the source of truth for what happened.

7.14.4. Extension points

Table 7.14 summarizes the main extension points left by the implementation. They correspond to specific places where the current architecture can grow.

Table 7.14.: Main extension points of the INF platform

Extension point	Current mechanism	How it can grow
New dataset	Processed Parquet files and seed scripts	Add dataset metadata, series definitions, and ingestion rules for a new target or context dataset.
New model	Forecasting adapter registry and model catalogue	Implement an adapter with the common input/output contract and expose it through a catalogue slug.
New context signal	MCP server tools and MCP client	Add a tool or processed signal source, then map it into backend context retrieval or exogenous features.
New metric	Metric computation and metric storage	Add a computed metric to the run execution path and expose it through comparison endpoints.
New dashboard	Frontend pages, API functions, and state hooks	Create a page or panel that consumes existing experiment, run, prediction, metric, or context endpoints.
Stronger monitoring	Health, drift endpoint, logs, and MLflow	Add scheduled checks, alerting, model/data drift dashboards, and deployment-level observability.
Production deployment	Docker Compose, gateway, environment settings	Harden secrets, CORS, TLS, backups, resource limits, migrations, and external service configuration.

These extension points allow the project to add new forecasting work, new context sources, or new user-facing views without changing the whole architecture at once.

7.14.5. Development limits

The implementation still has clear limits. The platform demonstrates an integrated architecture, but it does not include a full deployment pipeline, automated browser end-to-end tests, formal threat modelling, backup policy, or continuous monitoring stack. Some dependencies, such as external model APIs, local model packages, and news or context retrieval, can also change over time.

These limits also define a continuation path: converting the current reproducible development stack into a more hardened deployment with stronger security configuration, automated UI testing, data-version manifests, migrations, and continuous monitoring.

From a development perspective, the codebase is prepared for continuation. It separates forecasting artifacts from the application layer, exposes stable API workflows, stores run outputs, keeps contextual information traceable, and provides clear places for future datasets, models, metrics, and visualizations.

7.15. USER WORKFLOWS AND OPERATING GUIDE

This section connects the technical implementation with the practical operation of the system: how services are started, how data is made available, how experiments are created, and how results are inspected.

7.15.1. Initial execution workflow

The normal execution workflow starts from the repository root. First, the environment file is created from the example configuration and the required variables are reviewed. Then, Docker Compose starts the databases, backend, frontend, gateway, MLflow service, and MCP server as a coordinated stack.

After the services are running, the processed forecasting datasets must be loaded into PostgreSQL. The backend provides seed and ETL scripts for this purpose. In a complete local execution, the operator first starts the stack, then loads the IPC/CPI/HICP and exogenous datasets, and finally creates or seeds experiments and runs. This order matters because experiment creation depends on existing dataset, series, and model catalogue records.

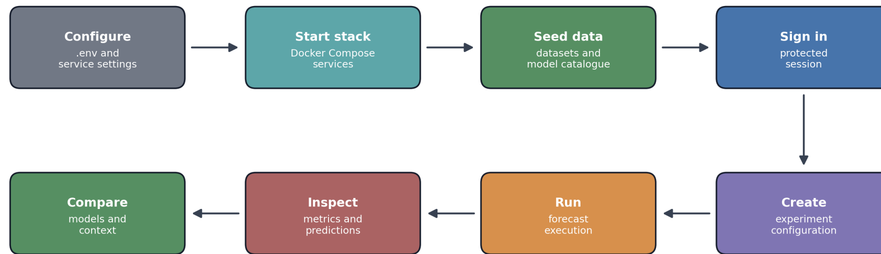


Figure 7.8.: Main user workflow from platform start-up to result inspection.

Figure 7.8 shows the operational sequence: configure environment, start services, seed datasets and catalogue entries, sign in, create an experiment, trigger a run, inspect predictions and metrics, compare models, and review contextual information.

7.15.2. Authentication and navigation

The frontend exposes public routes for landing, login, and sign-up. The rest of the application is protected. If a user is not authenticated, protected routes redirect to the login page. Once authenticated, the main navigation provides access to the experiment list, experiment creation, comparison dashboard, simulator, and current inflation-pulse view.

The navigation design reflects the main tasks of the platform. The user does not need to know the backend route names or database schema. Instead, the interface presents the platform as a set of workflows: manage experiments, run forecasts, compare results, explore what-if scenarios, and inspect live or recent contextual information.

7.15.3. Experiment workflow

The experiment workflow is the central user path. From the `New Experiment` page, the user selects a dataset, target series, model, forecast horizon, and whether MCP semantic context should be used. The form groups

models into statistical baselines and foundation models, so model families remain visible without exposing internal script names. The page also shows the JSON payload that will be sent to the backend, which supports transparency during evaluation.

Once the experiment is created, the user reaches the experiment detail page. This page summarizes the selected series, model, horizon, MCP flag, creation time, and run history. The user can trigger a new run from this page. The backend queues the run, updates its status, and the frontend refreshes the run list while the execution is pending or running.

This workflow transforms a forecasting configuration into a persistent application entity. The user can return to the experiment later, inspect previous runs, or launch another run with the same configuration.

7.15.4. Run inspection workflow

The run-detail page is the main result-inspection workflow. It shows the run status, start and finish timestamps, error message when applicable, metric cards, forecast chart, MCP context if it exists, and optional LLM-generated narration. The same page connects execution metadata, quantitative outputs, contextual evidence, and explanatory text.

The metrics shown in the current interface include MAE, RMSE, and MAPE. The forecast trajectory is displayed as a chart, while MCP-enabled runs can show contextual signal rows and sentiment information. The page also keeps failed runs visible, so failed model dependencies, missing API keys, unavailable MCP context, or unsupported model slugs can be inspected through the application.

The narration workflow is intentionally optional. It uses a local Ollama service to generate a readable explanation of a completed run. If the narration service is unavailable, the run itself remains valid because predictions and metrics are stored independently in PostgreSQL. This keeps explanation separate from the core forecasting result and from the CDIA interpretation of model performance.

7.15.5. Comparison workflow

The comparison dashboard allows the user to select multiple experiments for the same target series. This same-series restriction is a deliberate usability and validity decision: forecasts for different targets or scales should not be compared as if they were interchangeable. The dashboard then presents several comparison views, including forecast overlay, horizon-wise error behaviour, skill score, leaderboard, statistical significance, and

optional deep-dive charts.

This workflow makes model differences inspectable through an interactive comparison interface.

7.15.6. Simulation and context workflows

The simulator provides a what-if interface over selected datasets, series, and horizons. It lets the user modify macro or MCP-related signal values and observe how the counterfactual forecast changes. Its purpose is exploratory: to make the relation between contextual signals and forecast behaviour visible in the interface.

The `Today` view complements the simulator by showing a current inflation-pulse workflow. It retrieves recent inflation-related news, refreshes context when requested, and displays sentiment information derived through the MCP/news pipeline. This gives the user a separate entry point for understanding the textual context that motivates the MCP part of the project.

Together, these workflows let the user inspect completed experiments, explore contextual assumptions, and review recent signals from the same platform.

7.15.7. Workflow summary

Table 7.15 summarizes the main user-facing workflows and their technical support.

Table 7.15.: Main user-facing workflows of the INF platform

Workflow	Frontend entry point	Backend or service support
Sign in and access	Login and protected layout	Authentication routes, JWT tokens, current-user dependency, role checks.
Create experiment	`/experiments/new`	Dataset, series, model catalogue, and experiment-creation endpoint.
Trigger run	Experiment detail	Run endpoint, background task, adapter registry, PostgreSQL persistence.
Inspect run	`/runs/:id`	Run detail, predictions, metrics, MCP context, narration endpoint.
Compare models	`/compare`	Metrics comparison endpoint, selected experiments, same-series filtering.
Simulate context	`/simulator`	What-if setup endpoint, signal effects, counterfactual charting.
Review inflation pulse	`/today`	News endpoints, MCP/news pipeline, sentiment scoring, refresh workflow.

This operating view shows how the backend, frontend, databases, MCP server, MLflow service, adapters, and seed scripts support a coherent path from configuration to execution, storage, comparison, and contextual interpretation.

7.16. INTERFACE EVIDENCE

The INF platform includes several user-facing screens that should be represented visually in the final version of the thesis. Software user documentation should identify the user interface and the tasks that users perform with the software [33]. In this thesis, the figures provide evidence that the architecture described in the previous sections is exposed through concrete workflows.

The selected figures focus on the main implemented paths: experiment management, run execution, forecast inspection, model comparison, simulation, and contextual information.

7.16.1. Experiment management screens

The experiment-management evidence includes the experiment list and the new-experiment form. Together, these screens show how the user moves from an existing catalogue of experiments to the creation of a new forecasting configuration.

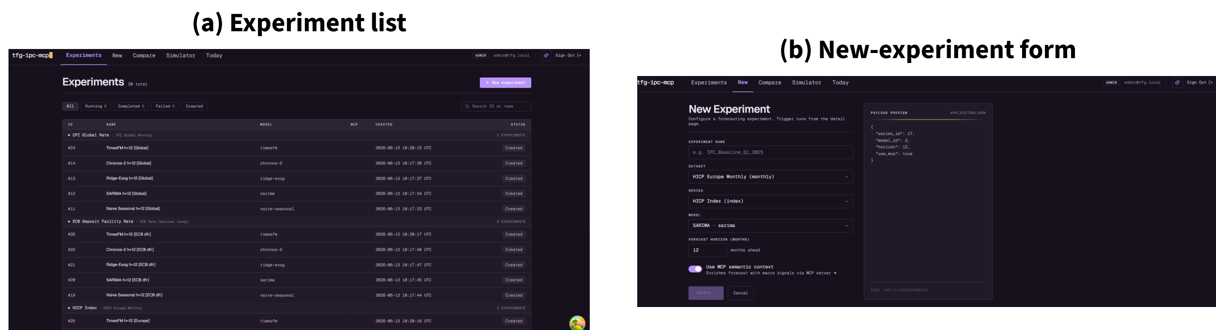


Figure 7.9.: Experiment list and new-experiment creation workflow.

Figure 7.9 shows how a forecasting configuration becomes an application entity through target-series selection, model-family selection, horizon configuration, and the MCP-context option.

7.16.2. Run inspection screens

The run-detail screen brings together experiment metadata, execution status, completed run records, and drift information.

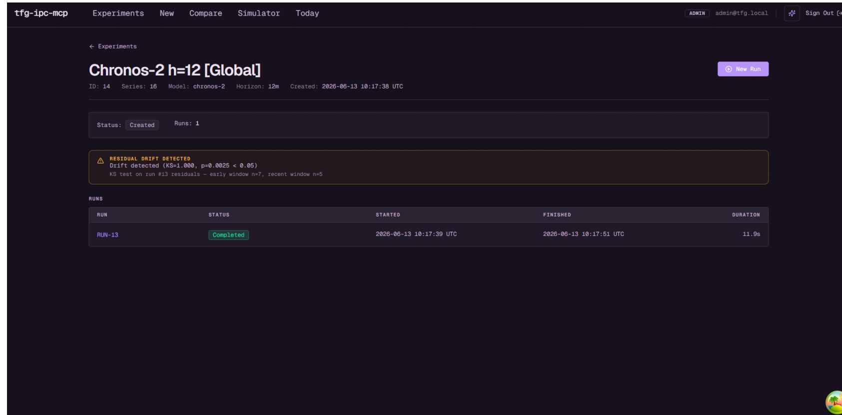


Figure 7.10.: Run-detail view with experiment metadata, completed run status, and drift warning.

Figure 7.10 demonstrates how the platform exposes a created experiment, the linked completed run, execution timestamps, duration, and the residual-drift warning returned by the drift endpoint.

7.16.3. Comparison dashboard screens

The comparison-dashboard screenshot shows the experiment selector, forecast overlay, and horizon-wise comparison views used to compare completed experiments.

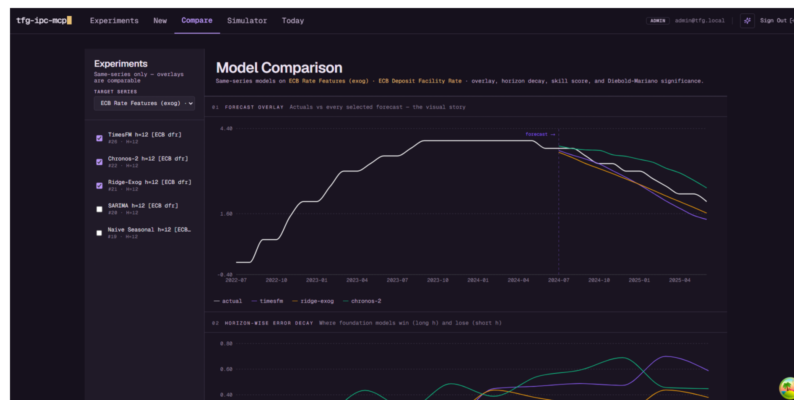


Figure 7.11.: Model comparison dashboard for same-series forecasting experiments.

Figure 7.11 shows that users can compare multiple completed experiments without manually opening result files. The same-series restriction remains part of the validity logic of the interface.

7.16.4. Simulation and context screens

The contextual views are represented by the what-if simulator and the inflation-pulse page. They show signal levers, baseline and counterfactual forecast curves, summary cards, recent headlines, and sentiment information from the running platform.

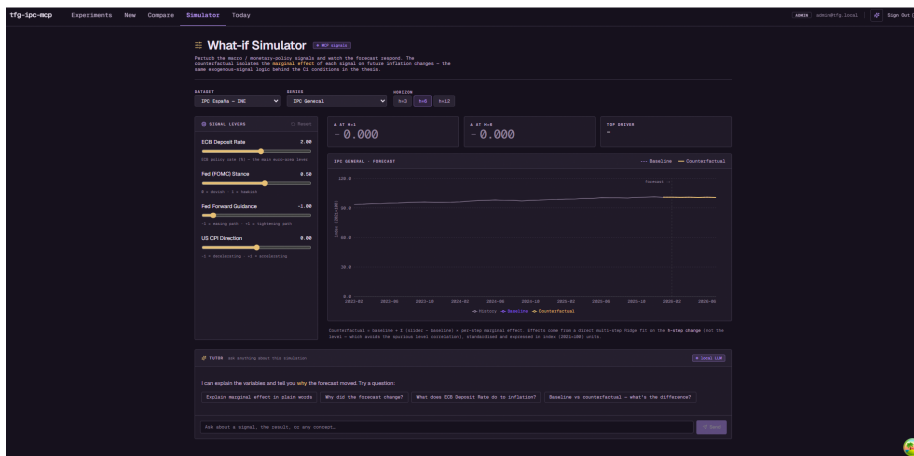


Figure 7.12.: What-if simulator with signal levers, horizon selection, forecast chart, and local assistant panel.

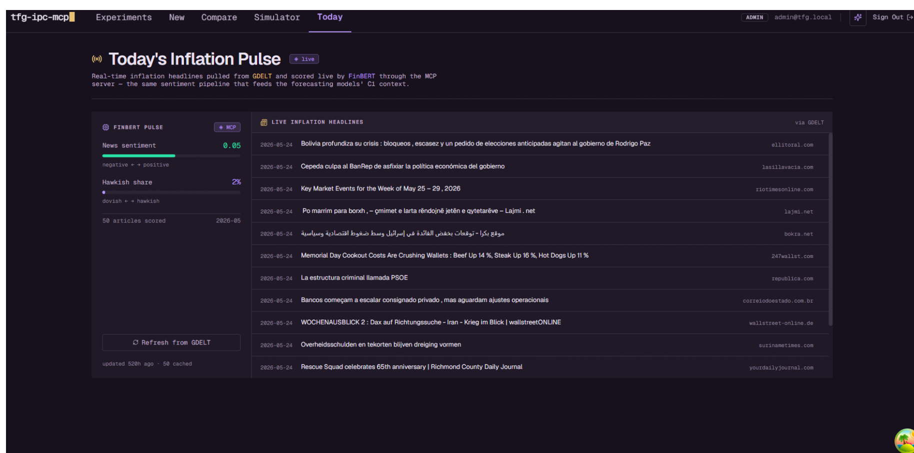


Figure 7.13.: Inflation-pulse page with FinBERT sentiment summary and live inflation-related headlines.

Figures 7.12 and 7.13 show how contextual information appears in the user interface. The simulator exposes controlled signal perturbations, while the inflation-pulse page displays news-derived sentiment and cached GDELT headlines.

7.16.5. Interface visual evidence status

Table 7.16 summarizes the interface visual evidence included in this chapter.

Table 7.16.: Interface visual evidence for the Development chapter

Figure	Screen or view	Evidence status
Figure 7.9	Experiment list and creation form	Real screenshots captured from the running INF platform.
Figure 7.10	Run page with experiment metadata, completed run status, and drift warning	Real screenshot captured from the running INF platform.
Figure 7.11	Same-series model comparison	Real screenshot captured from the running INF platform.
Figure 7.12	What-if simulator	Real screenshot captured from the running INF platform.
Figure 7.13	Inflation pulse	Real screenshot captured from the running INF platform.

7.17. REQUIREMENTS TRACEABILITY

The Development chapter began by defining the functional and non-functional requirements implemented in the platform. After presenting the main architecture and implementation areas, the chapter closes with a traceability view.

Requirements traceability is a common systems-engineering practice for connecting requirements with design decisions, implementation elements, and verification evidence [34]. Here, it shows how the implemented software answers the scope defined for the INF thesis and what remains as future improvement.

7.17.1. Functional coverage

Table 7.17 maps the functional requirements to the main implementation evidence described in this chapter. Requirements that belong to the same workflow are grouped, so the traceability view follows complete user paths rather than isolated endpoints.

Table 7.17.: Traceability of functional requirements

Requirement	Implemented support	Evidence in the chapter
FR-01	User access is supported through authentication routes, JWT-based sessions, protected frontend routes, and role-aware backend dependencies.	Backend/API section, frontend routing section, workflow summary, and backend integration tests.
FR-02 and FR-03	Dataset, target-series, and model catalogues are exposed through API routes and are used by the experiment-creation interface.	Data model section, seed-data description, experiment form workflow, and catalogue-related tests.
FR-04 and FR-05	Experiments can be created, listed, inspected, deleted, and executed. Forecast runs are launched through backend routes and executed through the adapter layer.	Experiment lifecycle, run execution flow, forecasting adapter section, and run lifecycle validation.
FR-06 and FR-07	Predictions and metrics are persisted after execution and reused by the run-detail and comparison views.	Persistence model, run-detail workflow, comparison dashboard, and metric-comparison validation.
FR-08 and FR-09	MCP context retrieval is implemented through the MCP server/client integration, MongoDB context storage, news endpoints, and contextual frontend panels.	MCP integration section, context persistence section, run-detail MCP panel, and inflation-pulse workflow.
FR-10	Simulation is exposed through a dedicated frontend view and backend support for what-if forecast exploration.	Simulator workflow and contextual exploration figure plan.
FR-11 and FR-12	Drift analysis and health checks provide operational visibility over the platform state.	Validation and testing section, health endpoint, drift-monitoring endpoint, and execution-environment discussion.

The table shows that the implemented system covers the functional scope as a connected application: configuration, execution, storage, inspection, comparison, and contextual interpretation.

7.17.2. Non-functional coverage

The non-functional requirements are less tied to a single screen or endpoint, but they determine whether the platform can be understood, reproduced, extended, and validated. Table 7.18 summarizes this coverage.

Table 7.18.: Traceability of non-functional requirements

Requirement	Implemented support	Evidence in the chapter
NFR-01	Modularity is supported by the separation between backend, frontend, MCP server, gateway, persistence services, shared code, and forecasting adapters.	Repository organization, general architecture, backend modules, and adapter registry.
NFR-02	Reproducibility is supported by Docker Compose, explicit configuration variables, seed scripts, persistent volumes, and MLflow tracking.	Execution environment and reproducibility section.
NFR-03	Extensibility is supported by model catalogues, adapter-based execution, MCP tools, and defined extension points for datasets, models, views, and services.	Forecasting adapter layer, MCP integration, and maintainability section.
NFR-04	Traceability is supported by persistent links between platform entities and contextual evidence.	Data model section, PostgreSQL schema discussion, MongoDB context storage, and run-detail workflow.
NFR-05	Maintainability is supported by clear module boundaries, typed API schemas, tests, service-level responsibilities, and explicit limits.	Backend/API section, validation and testing section, and maintainability discussion.
NFR-06	Robustness to external services is supported by health checks, status fields, error persistence, and graceful handling of optional MCP, MLflow, and LLM-related components.	Operational robustness section, execution environment, and validation tests.
NFR-07	Usability is supported by protected navigation, experiment forms, run pages, comparison dashboards, simulator views, and contextual screens.	Frontend application, user workflows, and interface evidence plan.
NFR-08	Validation support is provided through backend integration tests, drift analysis, metrics comparison, health endpoints, and persisted execution evidence.	Validation and testing section and forecasting-pipeline connection section.

The project defines service boundaries, persists experiment evidence, exposes results through repeatable workflows, and keeps the connection with contextual signals explicit.

7.17.3. Coverage limits

The traceability view also has limits. Backend integration tests and reproducible local execution do not replace a full security audit, a load test campaign, frontend end-to-end testing, or long-term monitoring in a deployed environment.

Some requirements are implemented as platform capabilities rather than industrial deployment capabilities. Authentication, authorization, health checks, experiment management, MCP integration, and result comparison are present and testable, while security hardening, observability, frontend automated testing, and operational monitoring remain future work.

8. ETHICAL ASSESSMENT

8.1. PURPOSE OF THE ASSESSMENT

This chapter analyses the ethical implications of the INF contribution: a software platform that organizes, executes, stores, and visualizes inflation-forecasting experiments. The assessment focuses on the engineering system, as defined in Chapter 3.

The platform is not a public decision system, a financial recommendation tool, or an automatic policy engine. Even so, it works with inflation forecasts, contextual economic signals, news-derived information, and AI-based model outputs. These elements can influence interpretation if they are presented without enough transparency, uncertainty, or context. The ethical discussion considers how the system communicates results, preserves traceability, limits overconfidence, and handles data and access control.

The European Commission's Ethics Guidelines for Trustworthy AI define trustworthy AI around requirements such as human agency and oversight, technical robustness, privacy and data governance, transparency, fairness, societal well-being, and accountability [35]. The OECD AI Principles also emphasize trustworthy AI, human rights, transparency, robustness, and accountability across the AI system lifecycle [36]. These frameworks apply to this project because the platform is an AI-supporting software system: it does not make decisions by itself, but it makes AI and forecasting outputs available to users.

8.2. RESPONSIBLE USE OF ECONOMIC FORECASTS

The main ethical risk of the project is not that the platform directly harms users through an automatic action. The greater risk is interpretative: a user may treat a forecast, comparison chart, or generated explanation as more certain than it really is. Inflation forecasts are uncertain by nature, and their errors can matter because inflation affects purchasing power, planning, contracts, wages, savings, and public debate.

This risk is mitigated by presenting forecasts as experiment outputs rather than as recommendations. Runs are linked to the platform entities and execution metadata defined in Chapter 7. Each result is a traceable technical artifact instead of an isolated number. The comparison dashboard also restricts model comparison to experiments from the same target series, reducing the chance of misleading comparisons between incompatible forecasts.

However, the system still requires careful communication. A graph with a smooth forecast curve can visually suggest certainty even when the model has limited predictive strength. Metric cards can also appear authoritative if the user does not understand the evaluation context. For this reason, the platform should maintain visible distinctions between observed values, predictions, counterfactual simulations, contextual indicators, and narrative explanations.

8.3. TRANSPARENCY AND TRACEABILITY

Transparency is supported by storing forecasts as inspectable platform records. Chapter 7 details that entity chain from configuration to output.

This traceability supports accountability in two ways. First, it allows a user or evaluator to reconstruct how a result was produced. Second, it makes limitations visible: failed runs, unavailable services, missing API keys, absent MCP context, or unsupported model slugs are represented as explicit system states rather than hidden failures.

The transparency requirement also applies to the MCP and news-related parts of the platform. Contextual signals and sentiment summaries should not be presented as neutral facts without explanation. They are derived from processed sources, retrieval rules, NLP models, and aggregation logic. The platform should show when context was retrieved, how many signal rows or articles were available, which run it belongs to, and whether the information was actually used by the selected adapter.

8.4. HUMAN OVERSIGHT AND AUTOMATION LIMITS

Human oversight is a recurring principle in AI governance. The EU AI Act, as the European Union's legal framework for AI, gives particular importance to trustworthy AI and risk management in systems that may affect people or fundamental rights [37]. This platform is not designed as a high-risk automated decision system, but the same principle is still useful: forecasts should support human analysis rather than replace it.

The implemented platform keeps a human-in-the-loop structure. Users define experiments, choose datasets and models, decide whether MCP context is enabled, trigger runs, inspect metrics, compare results, and interpret outputs. The system does not automatically issue decisions, policy actions, investments, or recommendations. It provides evidence and workflow support.

There are also automation limits that must remain clear. The simulator is exploratory and should not be inter-

preted as a causal economic model. MCP-derived corrections are technical mechanisms for testing contextual signals, not proof that text or macro signals always improve forecasts. Optional generated narration can help summarize a run, but it should not be treated as a source of truth independent from stored predictions, metrics, and execution metadata.

8.5. DATA GOVERNANCE AND PRIVACY

The platform mainly works with economic time-series data, processed macroeconomic signals, model outputs, and news-derived contextual information. These data are generally less sensitive than personal records, but data governance still matters. Datasets must be traceable to their source artifacts, transformations should be reproducible, and stored outputs should remain connected to the configuration that produced them.

The user-management layer introduces a small amount of personal information, mainly email addresses and authentication data. The backend addresses this at development level by storing passwords as hashes, using token-based authentication, protecting routes, and applying ownership checks to experiments and runs. Stronger governance would require secret rotation, audited access policies, data-retention rules, backups, logging policies, and a formal review of personal-data handling.

MongoDB context storage also requires care. News documents and sentiment-derived information may include public text, but the system should avoid storing unnecessary personal data, should keep contextual records connected to legitimate project purposes, and should document the origin and processing logic of the stored context.

8.6. FAIRNESS, BIAS, AND REPRESENTATIVENESS

Bias in this project is not limited to demographic fairness, because the system does not classify individuals. The relevant fairness issue is representativeness: which countries, institutions, languages, sources, and signals are better represented in the data, and which are not. The forecasting results already show that contextual signals may help some target series more than others. This has an ethical dimension because a platform can make weak or uneven evidence look equally reliable if the interface does not show its context.

For example, Global CPI and European HICP may benefit more from broad institutional and macroeconomic context, while Spain CPI may remain better captured by traditional seasonal dynamics. If the platform displayed all MCP-enabled forecasts as equally meaningful, it could encourage a false sense of generality. The interface

should preserve the distinction between target series, horizons, model families, and MCP conditions.

News-derived sentiment also introduces representativeness risks. News coverage is uneven across countries, languages, topics, and time periods. Sentiment models may reflect the training data and assumptions of their underlying NLP architecture. The platform should treat these values as contextual indicators, not as objective descriptions of the economy.

8.7. SECURITY AND MISUSE RISKS

The platform includes authentication, role separation, protected routes, and ownership checks, which are important for preventing unauthorized access to experiment records. These mechanisms also support ethical responsibility because they reduce the chance that users can inspect, modify, or delete records outside their permission level.

The main misuse risk is inappropriate use of outputs, in addition to unauthorized access. A user could export charts, forecasts, or generated explanations and present them outside the platform without the surrounding assumptions. This risk cannot be fully solved by code, but the system can reduce it by maintaining traceable metadata, clear labels, visible model names, timestamps, metric definitions, and contextual limitations.

External dependencies also create security and reliability considerations. The platform can depend on API keys, local model packages, MCP services, MLflow, databases, and optional language-model narration. The Development chapter already described graceful failure and health checks. Ethically, this matters because users should not receive silent or misleading outputs when part of the system has failed.

8.8. ENVIRONMENTAL AND RESOURCE CONSIDERATIONS

The project uses containerized services, databases, machine-learning packages, foundation-model adapters, and optional NLP models. This has a resource cost, especially when model weights are downloaded, containers are rebuilt, or experiments are repeated. Although the scale is limited, responsible engineering still requires attention to unnecessary computation.

The implementation reduces some waste by persisting run outputs, metrics, contextual records, MLflow information, and cached model files. Stored results can be inspected without rerunning every model, and persistent volumes avoid repeated downloads when the local environment is recreated. Future work could improve this further with clearer experiment manifests, data-version tracking, resource monitoring, and selective execution

policies for expensive models.

8.9. MITIGATION SUMMARY

Table 8.1 summarizes the main ethical risks and the mitigation mechanisms included or recommended for the platform.

Table 8.1.: Ethical risks and mitigation mechanisms

Risk	**Current mitigation**	**Future improvement**
Overconfidence in forecasts	Metrics, model names, horizons, run status, and comparison views are shown with each output.	Add clearer uncertainty bands, warning labels, and documentation for non-expert users.
Misinterpretation of simulations	Simulator is framed as exploratory and separated from completed forecast runs.	Add explicit simulator disclaimers and scenario metadata exports.
Loss of traceability	Platform entities, MCP context, and MLflow records are persisted.	Add data-version manifests and stronger audit trails.
Hidden service failure	Health checks, run status, failed states, and graceful degradation are implemented.	Add deployment monitoring, alerting, and frontend end-to-end tests.
Unauthorized access	JWT authentication, role checks, and ownership restrictions are implemented.	Harden secrets, CORS, token policies, and access logging before production use.
Bias in contextual signals	Context is stored and shown separately from core forecast outputs.	Document source coverage, language limitations, and sentiment-model assumptions.
Unnecessary computation	Outputs, metrics, MLflow logs, and model caches are persisted.	Add resource monitoring and execution-cost reporting.

8.10. RESIDUAL RESPONSIBILITY

The ethical position of the project is moderate and realistic. The platform does not remove uncertainty from inflation forecasting, and it should not be presented as an automatic decision system. It makes forecasting experiments more transparent, reproducible, inspectable, and operationally coherent.

The remaining responsibility lies in how the system is used and communicated. Forecasts must be interpreted as model outputs under specific assumptions. Contextual signals must be treated as additional evidence, not as causal proof. Generated explanations must remain secondary to stored data and metrics. If the platform were deployed beyond an academic setting, the next ethical step would be a stronger governance layer covering security, monitoring, privacy, user documentation, and formal risk assessment.

9. INCIDENTS

9.1. PURPOSE OF THE CHAPTER

This chapter records the main incidents and adjustments that appeared during the INF development. They are not presented as failures, but as practical situations that affected implementation, testing, integration, or documentation. The objective is to show how each situation was handled and how it influenced the final scope.

The incidents are connected with the nature of the project. The system combines backend and frontend services, databases, Docker orchestration, MCP context retrieval, forecasting adapters, external dependencies, and the forecasting pipeline. These connections created points where robustness or scope had to be clarified.

9.2. TIMEGPT API DEPENDENCY

One of the clearest incidents was the integration of TimeGPT. Unlike the local statistical adapters, TimeGPT depends on an external service and requires a `NIXTLA_API_KEY`. This meant that it could not be treated as a completely local model during development or testing.

The impact was practical. Runs using the TimeGPT adapter could fail when the API key was not configured, when the service was not reachable, or when external call limits made repeated testing inconvenient. This affected both the forecasting pipeline and the architecture work, because TimeGPT had to be selectable without making the whole backend depend on external configuration.

This dependency was handled explicitly. The backend adapter checks for the required API key and returns a controlled error when it is not available. The run lifecycle stores the failure as a failed run instead of hiding the problem or crashing the whole backend. Integration tests also check this behaviour, so missing external credentials become an expected graceful-failure path.

9.3. OPTIONAL MODEL PACKAGES

A related incident appeared with foundation-model adapters such as TimesFM and Chronos-2. These models depend on heavier packages and local model environments that are not always available in a lightweight development or test setup. Installing every model dependency in every environment would make testing slower

and less reproducible.

The impact was that the platform could not assume that all model families were executable at all times. If a missing package produced an uncontrolled exception, a single run could break the API workflow and make the frontend receive an unclear response.

The run execution path was designed around explicit status handling. Missing packages or unsupported model conditions are converted into failed run states with readable error messages. This allowed the backend integration suite to verify that the system remains stable even when some model families are unavailable.

9.4. MCP AND NEWS-SERVICE AVAILABILITY

The MCP integration also introduced an incident class related to service availability. MCP context retrieval depends on a separate MCP server, processed signal files, MongoDB news documents, and, for sentiment workflows, the FinBERT-related environment. During development, it was not realistic to assume that all these services and data sources would always be available.

The impact was that MCP-enabled workflows could not become a single point of failure. A forecast run should not fail only because contextual information cannot be retrieved, especially when the selected model can execute without that context. The same applies to the news and sentiment views: if no articles exist or the MCP/FinBERT path is unavailable, the user should receive an explicit unavailable state instead of a broken interface.

The solution was to separate optional context from the core run lifecycle. The backend logs MCP retrieval errors, stores context only when it is available, and allows compatible runs to continue without retrieved signals. The news endpoints also return controlled responses when sentiment cannot be computed. This incident influenced the final design of the robustness and validation sections.

9.5. TEST ISOLATION AND DATABASE STATE

Another relevant incident appeared during backend validation. The platform stores structured entities in PostgreSQL. If tests reused the same database state without isolation, results could depend on the order in which tests were executed.

The impact was typical of integration testing: one test could create records that affected the next one, making

failures difficult to interpret. This was especially sensitive for authentication, ownership checks, experiment creation, run execution, and metric comparison.

An isolated test setup was built to solve this problem. The integration fixtures create controlled users, datasets, model entries, and synthetic monthly observations. The relevant tables are reset between tests so that each test starts from a predictable state. This made the test suite more reliable and allowed API behaviour to be validated without depending on the full real forecasting datasets.

9.6. STATE-OF-THE-ART MATERIAL FOR THE INF THESIS

The INF background chapter initially had less software-engineering support than the forecasting literature base. The thesis needed a more specific base for MLOps, APIs, reproducibility, Docker, MCP integration, drift monitoring, and ML technical debt.

The impact was a documentation incident rather than a code incident. Without additional sources, the INF thesis risked describing the platform only from the local implementation, without enough connection to the state of the art in operational ML systems and software architecture.

To solve this, the `raw/articles-deep_search/` source bundle was created and ingested into the wiki. This added sources on MLOps, technical debt, REST/OpenAPI platforms, Docker deployment, MCP integration, and monitoring. Those sources supported the Background and Justification chapter with software-engineering evidence rather than relying only on the forecasting literature.

9.7. SCOPE ADJUSTMENT OF PRODUCTION FEATURES

A final incident was the need to control deployment-hardening scope. During development, several features naturally suggested further work: complete deployment hardening, frontend end-to-end tests, stronger monitoring, formal secret management, audit logs, backups, and a formal security review.

The impact was a scope risk. If all these features had been included as mandatory deliverables, the INF work would have expanded beyond the realistic duration of the TFG and would have reduced the time available for the core architecture, integration, and validation work.

A clear boundary was defined. The implemented platform includes a reproducible Docker Compose environment, backend integration tests, health checks, drift support, authentication, role checks, MCP integration, and

traceable run storage. Production hardening was left as future work. This kept the implemented scope at academic engineering-platform level instead of expanding it into a full industrial deployment.

9.8. INCIDENT SUMMARY

Table 9.1 summarizes the incidents and the corresponding responses.

Table 9.1.: Summary of incidents and responses

Incident	Impact	Response
TimeGPT API dependency	External API key and call constraints affected execution and testing.	Added explicit adapter error handling and graceful failed-run states.
Optional model packages	Some foundation-model environments were too heavy for all test setups.	Validated missing-dependency paths as controlled failures.
MCP/news availability	Context and sentiment services could be unavailable during execution.	Kept MCP context optional and returned controlled unavailable states.
Test database state	Shared database records could make tests order-dependent.	Added isolated fixtures and table reset logic for integration tests.
INF background sources	Initial software-engineering literature support was weaker than the forecasting literature support.	Created and ingested the INF deep-search source bundle.
Deployment-hardening expansion	Hardening features could exceed the implemented scope.	Documented them as future work while keeping the implemented platform focused.

Overall, these incidents did not change the central objective of the INF thesis. They helped refine it. The final platform gives priority to modularity, traceability, controlled failure, reproducible execution, and realistic scope management.

10. CONCLUSIONS AND FUTURE WORK

10.1. FINAL ASSESSMENT

This INF thesis has evaluated the engineering result of the INF platform. The final verdict is positive within the implemented project scope. The repository contains the multi-service stack and an adapter layer that exposes the forecasting logic through stable backend workflows.

As defined in Chapter 1, the platform contribution is operational. The entity-level evidence is assessed in the EQ1 verdict below.

The engineering result is bounded. The system is not a production-certified economic forecasting service. It is an academic software platform whose implementation evidence is visible in the repository: seven Docker Compose services, ten registered forecasting adapters, twelve functional requirements and eight non-functional requirements traced in Chapter 7, MLflow logging for run metadata and metrics, and a backend integration test suite that covers the main API workflows.

10.2. FULFILMENT OF OBJECTIVES

The general objective from Chapter 3 was to design and implement a modular INF platform for executing, storing, inspecting, and extending the inflation-forecasting workflow. The objective has been met at project scope because the implemented architecture connects the forecasting domain to backend routes, persistent entities, frontend workflows, adapter-based execution, MCP context retrieval, MLflow tracking, and Docker-based execution.

The requirement evidence is explicit. Chapter 7 defines twelve functional requirements, from `FR-01` user access to `FR-12` health and operational checks, and eight non-functional requirements, from `NFR-01` modularity to `NFR-08` validation support. The traceability matrix maps these requirements to implementation evidence such as authentication routes, dataset and model catalogues, experiment lifecycle routes, run execution, prediction and metric persistence, MCP context storage, simulator support, health checks, drift analysis, Docker Compose, MLflow tracking, and backend integration tests.

The backend validation evidence is verified in the Compose environment. The command `docker compose exec backend pytest -q` completed with `84 passed, 3 warnings in 128.84s (0:02:08)`. The inte-

gration tests cover authentication, datasets, experiments, health, metrics, runs, and users.

10.2.1. Engineering-question verdicts

The five engineering questions from Chapter 3 can be answered as follows:

1. **EQ1.** The platform preserves traceability by modelling the forecasting workflow as linked application entities. The evidence is the PostgreSQL schema for `datasets`, `series`, `observations`, `model_catalog`, `experiments`, `runs`, `predictions`, and `metrics`; the run execution path stores predictions and MAE, RMSE, and MAPE metrics for each completed run; and the traceability matrix maps `FR-04`–`FR-07` and `NFR-04` to those implementation elements.

2. **EQ2.** The system separates responsibilities through service and module boundaries rather than through a single notebook or script workflow. The evidence is the seven-service Docker Compose stack: `postgres`, `mongo`, `mlflow`, `backend`, `frontend`, `gateway`, and `mcp_server`; the backend package separation into `api`, `models`, `schemas`, `forecasting`, `mcp`, `db`, and `services`; and the traceability of `NFR-01` to those boundaries.

3. **EQ3.** MCP is integrated as a software capability, not as an INF claim about forecasting accuracy. The evidence is the MCP server, the backend MCP client, MongoDB storage for `mcp_contexts`, the `use_mcp` experiment flag, the `/runs/{run_id}/mcp-context` endpoint, the news endpoints, and the adapter execution logic that builds MCP exogenous data for foundation-model runs when context is enabled.

4. **EQ4.** Forecast results are inspectable through API and frontend workflows instead of requiring direct interaction with scripts. The evidence is the backend support for `/runs/{run_id}`, `/runs/{run_id}/predictions`, `/runs/{run_id}/metrics`, `/metrics/compare`, and `/metrics/dm-matrix`, together with frontend pages for experiment detail, run detail, comparison, today/news context, and simulation.

5. **EQ5.** The architecture is extensible at the adapter, service, and view levels, with limits that remain clear. The evidence is the registry of ten forecasting adapters: `naive-seasonal`, `arima`, `auto-arima`, `sarima`, `sarimax`, `ridge-exog`, `timesfm`, `chronos-2`, `timegpt`, and `ensemble-stack`; the active/inactive model catalogue field; the typed schemas; and the traceability of `NFR-03` to datasets, models, adapters, contextual signals, visualizations, and monitoring mechanisms.

10.3. ENGINEERING CONTRIBUTION

The INF part operationalizes the forecasting project as a software system. The evidence is the connection between the domain model, the execution layer, the persistence layer, the user interface, and the reproducible service stack.

First, the platform converts forecasting artifacts into application entities. The backend models define datasets and series, observations, model catalogue entries, experiments, runs, predictions, and metrics. These entities make the execution history inspectable after a run finishes instead of leaving the result only in local files.

Second, the platform exposes forecasting execution through adapters. The registry currently includes ten adapters, covering classical statistical models, exogenous variants, foundation-model interfaces, and an ensemble-stack adapter. This does not mean that every adapter has the same external dependency profile: TimeGPT requires an API key, some foundation-model adapters require heavier libraries, and the tests include cases for missing optional dependencies. The contribution is the registered adapter contract and the explicit failure handling around it.

Third, the platform adds operational evidence around runs. In `runs.py`, MLflow logs seven parameters for each tracked execution: `model_slug`, `horizon`, `use_mcp`, `series_id`, `experiment_id`, `run_id`, and `n_mcp_signals`. The same execution path logs the computed `mae`, `rmse`, and `mape` metrics. These records connect the run configuration with the observed outcome.

Fourth, the platform includes an initial monitoring-oriented endpoint. The drift endpoint computes residuals from the latest completed run of an experiment, splits them into an early 60% window and a recent 40% window, and applies a two-sample Kolmogorov–Smirnov test with an alpha threshold of 0.05. A real Compose-backed call to `GET /api/v1/drift?experiment_id=12` returned `experiment_id=12`, `run_id=11`, `drifted=true`, `p_value=0.0025`, `ks_statistic=1.0`, `n_early=7`, `n_recent=5`, and the message `Drift detected (KS=1.000, p=0.0025 < 0.05)`.

10.4. LIMITATIONS

The first limitation is validation environment dependency. The repository contains a backend integration suite with 84 tests that pass in the Docker Compose environment, but the suite requires the `postgres` service hostname and the project services configured for that environment. This means the validation evidence is reproducible through the documented Compose command, but not necessarily through an arbitrary local Python

environment.

The second limitation is production hardening. The platform includes JWT-based authentication, role-aware dependencies, health checks, status fields, persisted errors, and controlled handling for unsupported models or unavailable optional services. It does not include a full security audit, HTTPS deployment, backup policy, audit-log retention, secret-rotation procedure, load test campaign, or production monitoring process.

The third limitation concerns frontend validation. The repository contains user-facing pages for authentication, experiments, run inspection, comparison, news/context, and simulation, but the verified automated test evidence is concentrated in the backend integration suite. A complete product-grade validation plan would add frontend end-to-end tests for the main user workflows.

The fourth limitation is dependency variability. Some adapters depend on external APIs or optional libraries, MCP context depends on the availability and quality of contextual data, MLflow tracking is intentionally non-blocking, and news/sentiment workflows depend on external or cached sources. The platform records failures and keeps optional components separated, but these dependencies still affect reproducibility.

The fifth limitation is monitoring depth. The current drift endpoint is a first residual-distribution check, not a full model-governance subsystem. It does not implement scheduled checks, alert routing, retraining policies, approval workflows, or long-term data-quality monitoring.

10.5. FUTURE WORK

The following lines of future work follow directly from the evidence and limitations above.

10.5.1. Extend validation evidence

The backend integration suite and a real drift endpoint call have been recorded in this thesis. Future validation work should keep this evidence reproducible by storing execution logs with dates and service state, repeating the drift check across more completed runs and target series, and deciding which heavier Compose-backed checks can be automated without making the pipeline fragile.

10.5.2. Production hardening

A second line is to move the platform closer to a realistic deployment. This would include HTTPS configuration, stronger secret management, stricter role-based authorization review, audit logs, backup and recovery procedures, deployment documentation, and an API security review. These improvements would address risks that are outside the current implementation scope but necessary in a professional environment.

10.5.3. Continuous integration and automated quality

The repository already includes a continuous integration workflow with three jobs: backend Ruff linting, forecasting Ruff plus deterministic metric tests, and frontend dependency installation, linting, and build verification. Future work should therefore focus on the checks that are still missing from the automated quality layer: browser-based frontend end-to-end tests for experiment creation, run inspection, comparison dashboards, contextual panels, and simulator workflows; frontend and backend type checks where applicable; Docker image validation; and database migration checks.

10.5.4. Monitoring and model governance

The drift endpoint should be extended into a fuller monitoring layer. Future versions could add scheduled drift checks, alert thresholds, metric dashboards, data-quality reports, model-version policies, retraining triggers, and approval workflows for replacing or enabling forecasting models.

10.5.5. Improved MCP and contextual integration

The MCP layer can be extended with richer contextual tools, clearer provenance for news and institutional documents, better aggregation of sentiment and event signals, and stronger links between each retrieved context item and the forecast run that used it. This would strengthen traceability without making the INF thesis responsible for proving the predictive contribution of each signal.

10.5.6. More complete user experience

The frontend can be improved with richer chart interaction, export options, clearer status explanations, uncertainty visualizations, and more explicit distinctions between completed runs, failed runs, simulations, and contextual explanations. These changes would reduce the risk that users overinterpret a forecast or confuse

exploratory simulation with validated model output.

10.5.7. Extension of models and datasets

The adapter registry and catalogue structure make it possible to add new models and target series. Future work could add further statistical baselines, additional foundation-model interfaces, richer ensemble mechanisms, new inflation indicators, or other macroeconomic variables. Each extension should be accompanied by the same traceability and validation evidence used in the current platform.

10.6. CLOSING STATEMENT

The INF contribution is supported by seven orchestrated services, ten forecasting adapters, persistent experiment entities, contextual MCP integration, MLflow run logging, backend validation tests, frontend inspection workflows, and an explicit traceability matrix. These elements support the final verdict that the project has produced a working platform for executing, storing, comparing, and inspecting inflation-forecasting workflows.

The remaining work is also clear. The measured test and drift evidence should be preserved as reproducible validation records, while the next technical steps should continue toward stronger deployment, monitoring, frontend validation, automated type and image checks, migration validation, and model-governance mechanisms.

11. BIBLIOGRAPHY

[1] Y. Liang, H. Wen, Y. Nie, Y. Jiang, M. Jin, D. Song, and S. Pan, “Foundation Models for Time Series Analysis: A Tutorial and Survey,” *arXiv preprint arXiv:2403.14735*, 2024, available: https://arxiv.org/abs/2403.14735.

[2] L. J. Tashman, “Out-of-Sample Tests of Forecasting Accuracy: An Analysis and Review,” *International Journal of Forecasting*, vol. 16, no. 4, pp. 437–450, 2000, available: https://doi.org/10.1016/S0169-2070(00)00065-0.

[3] R. J. Hyndman and A. B. Koehler, “Another Look at Measures of Forecast Accuracy,” *International Journal of Forecasting*, vol. 22, no. 4, pp. 679–688, 2006, available: https://doi.org/10.1016/j.ijforecast.2006.03.001.

[4] F. X. Diebold and R. S. Mariano, “Comparing Predictive Accuracy,” *Journal of Business & Economic Statistics*, vol. 13, no. 3, pp. 253–263, 1995, available: https://doi.org/10.1080/07350015.1995.10524599.

[5] D. Kreuzberger, N. Kuehl, and S. Hirschl, “Machine Learning Operations (MLOps): Overview, Definition, and Architecture,” *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023, available: https://doi.org/10.1109/ACCESS.2023.3262138.

[6] Google Cloud, “MLOps: Continuous Delivery and Automation Pipelines in Machine Learning,” https://docs.cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning, 2024, accessed: 2026-06-09.

[7] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, “Hidden Technical Debt in Machine Learning Systems,” in *Advances in Neural Information Processing Systems*, 2015, pp. 2503–2511, available: https://papers.neurips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.

[8] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000, available: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf.

[9] OpenAPI Initiative, “OpenAPI Specification,” https://swagger.io/specification/, 2025, accessed: 2026-06-09.

[10] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie,

M. Parkhe, F. Xie, and C. Zumar, "Accelerating the Machine Learning Lifecycle with MLflow," *IEEE Data Engineering Bulletin*, vol. 41, no. 4, pp. 39–45, 2018, available: https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf.

[11] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, no. 239, p. 2, 2014, available: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[12] Docker, "Docker Compose Documentation," https://docs.docker.com/compose/, 2026, accessed: 2026-06-09.

[13] Model Context Protocol, "Model Context Protocol Specification," https://modelcontextprotocol.io/specification/2025-03-26, 2025, accessed: 2026-06-09.

[14] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A Survey on Concept Drift Adaptation," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–37, 2014, available: https://doi.org/10.1145/2523813.

[15] A. Mallick, K. Hsieh, B. Arzani, and G. Joshi, "Matchmaker: Data Drift Mitigation in Machine Learning for Large-Scale Systems," in *Proceedings of Machine Learning and Systems*, vol. 4, 2022, available: https://proceedings.mlsys.org/paper_files/paper/2022/hash/069a002768bcb31509d4901961f23b3c-Abstract.html.

[16] Universidad Internacional de La Rioja, "Pedro Gómez Tejerina," https://www.unir.net/profesores/pedro-gomez-tejerina/, 2026, accessed: 2026-06-12.

[17] FastAPI, "FastAPI Documentation," https://fastapi.tiangolo.com/, 2026, accessed: 2026-06-14.

[18] SQLAlchemy, "SQLAlchemy Documentation," https://docs.sqlalchemy.org/, 2026, accessed: 2026-06-14.

[19] React, "React Documentation," https://react.dev/, 2026, accessed: 2026-06-10.

[20] Vite, "Vite Documentation," https://vite.dev/, 2026, accessed: 2026-06-10.

[21] React Router, "React Router Documentation," https://reactrouter.com/, 2026, accessed: 2026-06-10.

[22] TanStack, "TanStack Query React Documentation," https://tanstack.com/query/v5/docs/framework/react/overview, 2026, accessed: 2026-06-10.

[23] Recharts, "Recharts Documentation," https://recharts.org/, 2026, accessed: 2026-06-10.

[24] Docker, "Docker Documentation," https://docs.docker.com/, 2026, accessed: 2026-06-14.

[25] Astral, "uv Documentation," https://docs.astral.sh/uv/, 2026, accessed: 2026-06-14.

[26] NGINX, "nginx Documentation," https://nginx.org/en/docs/, 2026, accessed: 2026-06-14.

[27] MLflow, "MLflow Tracking Documentation," https://mlflow.org/docs/latest/ml/tracking/, 2026, accessed: 2026-06-10.

[28] pytest, "pytest Documentation," https://docs.pytest.org/en/stable/, 2026, accessed: 2026-06-14.

[29] ——, "How to Use Fixtures," https://docs.pytest.org/en/stable/how-to/fixtures.html, 2026, accessed: 2026-06-11.

[30] FastAPI, "Testing," https://fastapi.tiangolo.com/tutorial/testing/, 2026, accessed: 2026-06-11.

[31] pandas, "pandas.read_parquet Documentation," https://pandas.pydata.org/docs/reference/api/pandas.read_parquet.html, 2026, accessed: 2026-06-11.

[32] OWASP Foundation, "OWASP API Security Top 10 2023," https://owasp.org/www-project-api-security/, 2023, accessed: 2026-06-11.

[33] IEEE, "IEEE 26514-2010: Systems and Software Engineering – Requirements for Designers and Developers of User Documentation," https://standards.ieee.org/ieee/26514/4990/, 2010, accessed: 2026-06-11.

[34] NASA, "NASA Systems Engineering Handbook," https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf, 2016, accessed: 2026-06-11.

[35] European Commission High-Level Expert Group on Artificial Intelligence, "Ethics Guidelines for Trustworthy AI," https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai, 2019, accessed: 2026-06-11.

[36] OECD, "OECD AI Principles," https://www.oecd.org/en/topics/sub-issues/ai-principles.html, 2024, accessed: 2026-06-11.

[37] European Commission, "AI Act: Regulatory Framework for Artificial Intelligence," https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai, 2024, accessed: 2026-06-11.

[38] University of Deusto Artificial Intelligence Committee, "Use of Artificial Intelligence," https://ai-label.org,

2026, guidance sheet provided for students; accessed: 2026-06-12.

[39] Anthropic, “Claude,” https://claude.ai/, 2026, accessed: 2026-06-15.

[40] ——, “Claude Code,” https://code.claude.com/, 2026, aI coding assistant; accessed: 2026-06-15.

[41] Google, “Stitch,” https://stitch.withgoogle.com/, 2026, aI-assisted user-interface design tool; accessed: 2026-06-15.

DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

ACRONYMS AND ABBREVIATIONS

AI Artificial Intelligence. In this thesis, the term is used mainly for the forecasting and contextual-signal components that the software platform integrates and exposes.

API Application Programming Interface. In this project, it mainly refers to the REST API implemented with FastAPI and consumed by the frontend and gateway layers.

ARIMA Autoregressive Integrated Moving Average. Classical statistical time-series model family used in the forecasting part of the integrated project and exposed through the platform catalogue.

ASGI Asynchronous Server Gateway Interface. Python interface used by modern asynchronous web frameworks and servers, including the FastAPI backend stack.

AutoARIMA Automatic ARIMA. Procedure that selects an ARIMA specification automatically according to statistical criteria and validation logic.

BBVA Banco Bilbao Vizcaya Argentaria. Banking group mentioned in the project motivation through the professional profile cited in Chapter 2.

C0 Baseline forecasting condition that uses only the historical target series.

C1 Contextual forecasting condition that introduces external information, such as institutional, macroeconomic, energy, uncertainty, or text-derived signals.

CDIA Ciencia de Datos e Inteligencia Artificial. Degree area focused on data science and artificial intelligence.

CI Continuous Integration. Automated validation process for code changes, usually involving tests, builds, and quality checks.

CORS Cross-Origin Resource Sharing. Browser security mechanism configured so that the frontend can communicate with backend services under controlled origins.

CPI Consumer Price Index. General inflation indicator. In the project, it appears mainly through Spain CPI and Global CPI series.

DDME Data-Driven Model Engineering. Methodological idea used in the INF thesis to connect data-science artifacts with engineering workflows and software system design.

ECB European Central Bank. Institution whose rates, releases, and data sources are relevant to the contextual layer used by the broader forecasting project.

ETL Extract, Transform, Load. Data-processing workflow used to collect, transform, and prepare data before it is consumed by forecasting or platform components.

EQ Engineering Question. Question used in the INF thesis to evaluate whether the implemented platform answers the main software-engineering concerns.

FOMC Federal Open Market Committee. United States monetary-policy body whose communications and calendars may appear in contextual or institutional signal layers.

FR Functional Requirement. Requirement that describes a behaviour or capability that the platform must provide.

GDELT Global Database of Events, Language and Tone. External news and event-data source referenced by contextual and semantic signal workflows.

HICP Harmonised Index of Consumer Prices. European inflation indicator used for comparable inflation measurement across European countries.

HTTP Hypertext Transfer Protocol. Application-layer protocol used by the web platform for communication between browser, gateway, backend, and services.

HTTPS Hypertext Transfer Protocol Secure. Encrypted version of HTTP expected for secure deployment of the platform outside a local development environment.

INF Informatics or Computer Engineering degree area of the integrated TFG. The INF thesis focuses on architecture, implementation, integration, deployment, validation, and maintainability.

IPC *Indice de Precios de Consumo*. Spanish term for Consumer Price Index and part of the project name.

IPC-MCP Project shorthand that combines the inflation-forecasting target, IPC, with the Model Context Protocol integration used for contextual economic signals.

JSON JavaScript Object Notation. Structured data format used for API payloads, metrics, model traces, configuration fragments, and intermediate outputs.

JWT JSON Web Token. Token format used to represent authenticated sessions or authorization claims in the platform.

LLM Large Language Model. AI model family used in the broader context of contextual retrieval and economic text interpretation.

LSTM Long Short-Term Memory. Recurrent neural-network architecture used as a deep-learning baseline in the forecasting part of the integrated project.

MAE Mean Absolute Error. Forecasting error metric referenced by the platform.

MAPE Mean Absolute Percentage Error. Forecasting metric that expresses absolute error as a percentage of the observed value.

MASE Mean Absolute Scaled Error. Normalized forecasting metric used to compare a model against a baseline scale.

MCP Model Context Protocol. In this project, MCP refers to the software and signal-access layer used to collect, organize, and expose contextual information for forecasting workflows.

ML Machine Learning. General field related to the forecasting models, operational workflows, and model-management concerns integrated by the platform.

MLflow Machine-learning lifecycle and experiment-tracking platform included in the service stack to support run traceability and future model-management workflows.

MLOps Machine Learning Operations. Engineering discipline concerned with deploying, validating, monitoring, and maintaining machine-learning systems.

MVP Minimum Viable Product. Milestone label used in the project plan to group deliverables into coherent increments.

N-BEATS Neural Basis Expansion Analysis for Interpretable Time Series Forecasting. Deep-learning forecasting architecture referenced as part of the forecasting model families.

N-HiTS Neural Hierarchical Interpolation for Time Series Forecasting. Deep-learning forecasting architecture referenced as part of the forecasting model families.

NFR Non-Functional Requirement. Requirement that describes qualities or constraints of the system, such as security, maintainability, traceability, reproducibility, or performance.

NLP Natural Language Processing. Area related to the treatment of textual economic context and semantic signals.

Nixtla Provider of the TimeGPT forecasting service referenced by the `NIXTLA_API_KEY` environment variable.

OECD Organisation for Economic Co-operation and Development. International organization cited for AI-principle guidance.

Ollama Local model-serving tool used by the optional narrative and simulator assistant endpoints.

Anthropic Provider referenced by the optional `ANTHROPIC_API_KEY` environment variable in the example configuration.

OWASP Open Worldwide Application Security Project. Reference organization for application-security practices used as a practical baseline for web-platform risk discussion.

PFG Proyecto Fin de Grado. Final Degree Project.

REST Representational State Transfer. Architectural style used by the backend API to expose resources and operations through HTTP endpoints.

RMSE Root Mean Squared Error. Forecasting error metric that gives more weight to large forecast errors than MAE.

SARIMA Seasonal Autoregressive Integrated Moving Average. ARIMA extension that includes seasonal dynamics.

SARIMAX Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors. SARIMA extension that incorporates external variables.

SQL Structured Query Language. Language family used to query and manage relational databases such as PostgreSQL.

SSE Server-Sent Events. HTTP-based mechanism that allows a server to push one-way event updates to a browser client.

TFG Trabajo Fin de Grado. Final degree project.

TimeGPT Foundation time-series forecasting model accessed through an external API.

TimesFM Foundation time-series forecasting model used in the forecasting study and exposed through the platform adapter catalogue.

TLS Transport Layer Security. Cryptographic protocol used to secure network communication in production deployments.

US User Story. Planning label used to identify user-centred development tasks in the work plan.

UTF-8 Unicode Transformation Format 8-bit. Character encoding used by the streaming assistant response described in the backend implementation.

UI User Interface. Visual and interactive layer through which users operate the platform.

URI Uniform Resource Identifier. String that identifies a resource, commonly used in API and web-routing contexts.

URL Uniform Resource Locator. Web address used to locate pages, services, endpoints, and external resources.

UX User Experience. Quality of the interaction between users and the platform workflows.

KEY DEFINITIONS

Adapter Software component that exposes a forecasting model through a common execution interface, independently of the internal implementation of that model.

Backend Server-side part of the platform. It contains the API, business logic, database access, forecasting orchestration, authentication, and integration with external or auxiliary services.

Contextual signal Economic, institutional, news-based, or sentiment-related information used to complement the time-series forecasting workflow.

Docker Containerization technology used to package and run services in a reproducible environment.

Experiment User-defined forecasting configuration that selects a dataset, target series, model, horizon, and optional context usage.

FastAPI Python framework used to implement the backend API of the platform.

Forecast run Concrete execution of an experiment. A run stores status, timestamps, predictions, metrics, errors, and optional contextual evidence.

Frontend User-facing part of the platform. It presents forms, dashboards, charts, comparisons, simulations, and contextual views through the browser.

Gateway Entry-point service used to route traffic toward the platform services, especially in a deployment-oriented setup.

Model catalogue Registry of forecasting models exposed by the platform, including their identifiers, family, description, and execution availability.

MongoDB Document database used for contextual and news-related information.

Pipeline Ordered sequence of data processing, modelling, evaluation, or integration steps. In this project, the term appears both in the forecasting workflow and in the software integration workflow.

PostgreSQL Relational database used for structured application entities such as users, datasets, experiments, runs, predictions, and metrics.

Reproducibility Capacity to execute, inspect, and continue the project with controlled configuration, structured outputs, and a defined multi-service environment.

Traceability Capacity to follow the relationship between datasets, series, models, experiments, runs, predictions, metrics, and contextual evidence.

Univariate forecast Forecast that uses only the historical target series and no external contextual variables.

A. APPENDICES

The appendices collect supporting material that is useful for understanding, running, or completing the INF platform thesis, but that would interrupt the main development narrative if included in Chapter 7. They are written as a technical reminder for the final review of the project.

A.1. REPOSITORY AND SERVICE MAP

The complete GitHub repository of this project is available at DiegoRamirezLacalle/tfg-ipc-mcp. Table A.1 summarizes the main repository areas used by the INF thesis.

Table A.1.: Repository areas relevant to the INF thesis

Path or file	**Role**
Backend	FastAPI backend at `tfg-arquitectura/backend/`: API routes, schemas, models, services, forecasting adapters, database access, and tests.
Frontend	React frontend at `tfg-arquitectura/frontend/`: pages, reusable components, hooks, API client logic, and user-facing workflows.
MCP server	Service at `tfg-arquitectura/mcp_server/` used to expose contextual tools and macro/news-related information.
Gateway	Gateway service at `tfg-arquitectura/gateway/` and Nginx configuration for routing platform traffic.
Database initialization	Files at `tfg-arquitectura/db/`, especially for MongoDB context/news storage.
`tfg-forecasting/`	Forecasting research pipeline and processed artifacts used as the analytical base of the integrated project.
`shared/`	Shared constants, data-loading utilities, metrics, and support code used across the repository.
`docker-compose.yml`	Multi-service execution definition for backend, frontend, gateway, PostgreSQL, MongoDB, MLflow, and MCP server.

Table A.2 summarizes the service stack.

Table A.2.: Service stack used by the platform

Service	Default port	Responsibility
Backend	8000	API, authentication, experiment management, run execution, metrics, MCP communication, drift, and simulation support.
Frontend	3000	Browser interface for platform workflows.
Gateway	80	Deployment-oriented entry point and traffic routing.
PostgreSQL	5432	Structured relational storage for platform entities.
MongoDB	27017	Contextual and news-related document storage.
MLflow	5000	Experiment-tracking support and future model-management base.
MCP server	8080	Contextual tool and signal retrieval service.

A.2. EXECUTION REMINDER

The platform is designed to be executed through Docker Compose. Before running it, the example environment file must be copied and adapted to the local machine. Development credentials and example secrets must not be reused in a production deployment.

```
cp .env.example .env
docker compose up --build
```

Pseudocode A.1: Basic local execution reminder

The most relevant environment variables are summarized in Table A.3.

Table A.3.: Relevant configuration variables

Variable	Purpose
`POSTGRES_*`	PostgreSQL connection settings: user, password, database, host, and port.
`MONGO_URI`	MongoDB connection string for context and news storage.
`NIXTLA_API_KEY`	Optional TimeGPT credential; missing keys produce controlled failures.
`GDELT_API_KEY`	Optional credential for external news/data workflows.
`ANTHROPIC_API_KEY`	Optional key for LLM-assisted context or narration workflows.
`JWT_*`	Authentication-token settings; development values must be changed.
`ADMIN_*`	Initial administrator seed credentials from the example file.
Backend/MCP ports	Local port configuration for backend and MCP services.

A.3. API ENDPOINT FAMILIES

Table A.4 groups the main API areas implemented by the backend.

Table A.4.: Main backend endpoint families

Endpoint family	Main purpose
`/auth`	Signup, login, and current-user retrieval.
`/users`	Role-management operations restricted to authorized users.
`/datasets`, `/series`, `/models`	Dataset, target-series, observation, and model-catalogue access.
`/experiments`	Experiment creation, listing, detail retrieval, run listing, and deletion.
`/runs`	Run execution, run detail, predictions, metrics, MCP context, and narration support.
`/metrics`	Comparison-oriented metric endpoints for completed runs.
`/news`	News refresh, current news retrieval, and sentiment-related responses.
`/drift`	Initial drift-analysis endpoint based on residual-distribution comparison.
`/whatif`	Setup data for simulation and what-if exploration.
`/assistant`	Assistant-oriented simulator chat endpoint.
`/health`	Platform health status.

A.4. VALIDATION REMINDER

The backend integration tests provide the main automated validation evidence for the INF thesis. Table A.5 summarizes the tested areas.

Table A.5.: Backend validation areas

Test area	**Validation focus**
Authentication	Signup, duplicate signup, password validation, login, token-protected access, and public-role restrictions.
Datasets and models	Dataset listing, series retrieval, observation pagination, missing-resource responses, and model catalogue access.
Experiments	Experiment creation, listing, detail retrieval, ownership rules, deletion, and invalid input handling.
Runs	Run triggering, completed and failed states, predictions, metrics, unsupported models, missing packages, TimeGPT missing key, and MCP-unavailable execution.
Metrics	Run comparison, ordering, deduplication, access control, empty responses, and response shape.
Users	Administrator role changes, forbidden role changes, missing users, and self-demotion protection.
Health	Availability and shape of the health response.

A basic backend test execution reminder is shown in Listing A.2. The exact command may be adapted depending on whether the tests are executed locally, inside a container, or through a future continuous-integration pipeline.

```
cd tfg-arquitectura/backend
pytest tests
```

Pseudocode A.2: Backend test execution reminder

A.5. DECLARATION OF ARTIFICIAL INTELLIGENCE USE

This appendix declares the use of generative artificial intelligence tools during the development and writing of this INF thesis. It follows the transparency criterion recommended by the University of Deusto guidance on the use of artificial intelligence in academic work [38]. The objective of this declaration is not to reproduce the prompts used during the project, but to state which tools were used, in which stages, and for what purpose.

I developed the project and wrote the thesis, using the artificial intelligence tools described below only as support for research organization, code assistance, debugging, figure preparation, and writing revision. Their outputs were not accepted automatically. I reviewed them, contrasted them with reliable sources, checked them against the actual repository and thesis files when relevant, and rewrote them where necessary to preserve the accuracy and authorship of the work.

Table A.6.: Use of artificial intelligence tools during the INF project

Tool	**Stage of the project**	**Purpose of use**
Claude [39]	Background, state-of-the-art analysis, and technical understanding	I used Claude to support the search, organization, and synthesis of information about MLOps, API-based platforms, Docker-based execution, experiment tracking, Model Context Protocol integration, and monitoring concepts. I checked the resulting suggestions against papers, official documentation, project files, and the bibliography before incorporating any content into the thesis.
Claude Code [40]	Software development, debugging, and documentation support	I used Claude Code as a coding assistant while reviewing and improving parts of the repository and thesis assets, especially backend/frontend organization, test interpretation, LaTeX consistency, figure generation, and command-based validation. I inspected, adapted, executed, and verified code or documentation suggestions before keeping them.
Google Stitch [41]	Frontend ideation and interface-design support	I used Stitch to support early frontend interface exploration, visual layout alternatives, and screen-level design ideas for the React application. The generated proposals were treated as design assistance only; I adapted the final interface manually in the repository, checked it against the implemented workflows, and validated the resulting screens through the running INF platform.
Claude and Claude Code writing assistance [39, 40]	Draft revision and final document preparation	I wrote the thesis text and used these tools to review clarity, grammar, structure, and professional tone in English paragraphs, tables, captions, appendix material, and explanatory sections. I made the final wording decisions and edited the text so that it reflected the work actually carried out in the INF contribution.

In all cases, I used the tools to support learning and productivity rather than to replace the core engineering work. I decided the architecture, repository organization, platform scope, validation evidence, and final conclusions. Software evidence was taken from the implemented repository, local command outputs, generated figures, tests, and LaTeX compilation results, not from artificial intelligence outputs. When I used AI assistance for technical explanations, I verified the content against primary sources, official documentation, or the actual project implementation. When I used it for code or figure generation, I reviewed the files and executed the corresponding commands before accepting the result.

The main safeguards applied during the project were the following:

- AI-generated explanations were checked against papers, official documentation, or project files before being used.
- Architectural claims, endpoint descriptions, service maps, and validation summaries were grounded in the repository and in the implemented platform structure.
- Code and figure-generation suggestions were executed, inspected, and corrected before being accepted.
- Textual revisions were used to improve clarity and correctness, not to change the meaning of the work or introduce unsupported claims.
- I retained responsibility for the final content, interpretation, and academic validity of the thesis.

The use of artificial intelligence in this thesis was limited, transparent, and supervised. It contributed to technical organization, software assistance, figure preparation, and language quality, while the final evidence, reasoning, and conclusions remained grounded in the implemented project and in verified sources.

A.6. FINAL EVIDENCE CHECKLIST

The main explanatory diagrams and interface screenshots have been added to the thesis with a consistent project visual style. The final interface evidence now includes experiment management, run detail, comparison, simulator, and inflation-pulse captures from the running INF platform.

Table A.7.: Final interface evidence status

Screenshot or asset	Current status
Experiment management screenshots	Attached in Figure 7.9 from the running INF platform.
Run-detail screenshot	Attached in Figure 7.10 from the running INF platform.
Comparison-dashboard screenshot	Attached in Figure 7.11 from the running INF platform.
Simulator screenshot	Attached in Figure 7.12 from the running INF platform.
Inflation-pulse screenshot	Attached in Figure 7.13 from the running INF platform.

Before final submission, the thesis should also be reviewed for final title consistency, cover pages, signed documents if required, bibliography consistency, and figure numbering.