egalLens / LegalLens C4 Model

Introduction

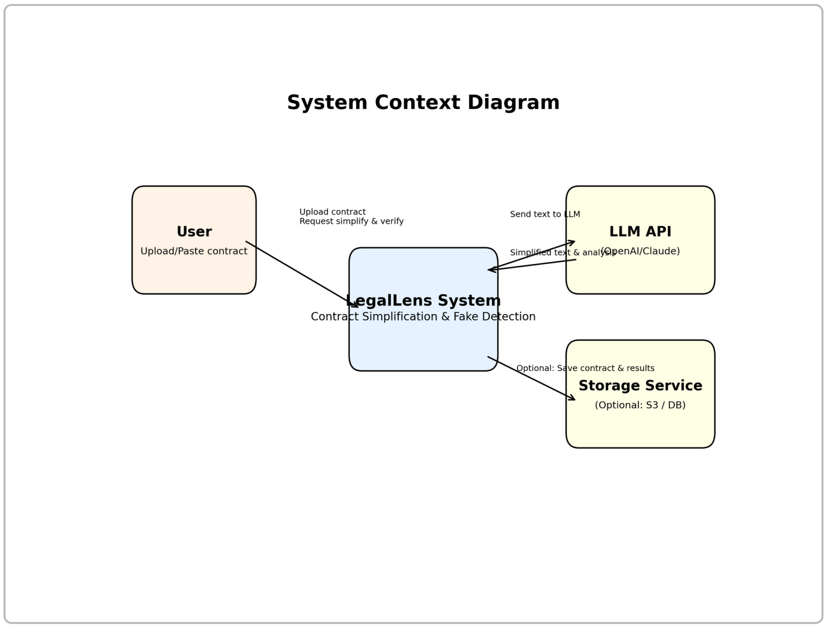
The C4 model provides a hierarchical way to visualise and communicate the architecture of a software system [1]. It defines four levels:

- **System context** – a high-level view showing the software system in scope, the people who use it, and other external systems [2].
- **Container** – zooms in to show the high-level technology and responsibilities of the major software "containers" (applications, services, databases) within the system [3].
- **Component** – decomposes a container to show the components within it and their responsibilities [4].
- **Code** – zooms further into a component to illustrate its implementation using class or entity diagrams [5].

The following C4 model diagrams describe **egalLens (LegalLens)** – a contract simplification and fake-detection application that allows users to upload contracts, simplifies legal terms using an LLM, and highlights potential fraud.

System Context Diagram

The system context diagram (Level 1) highlights how the LegalLens system fits into its environment. It focuses on people (the user) and external systems rather than implementation technology [2]. The user uploads a contract via a web interface. The LegalLens system may send the extracted text to an external LLM API (e.g. OpenAI GPT-4 or Claude) to simplify the language and perform fraud analysis. Optionally, results can be persisted to a storage service.

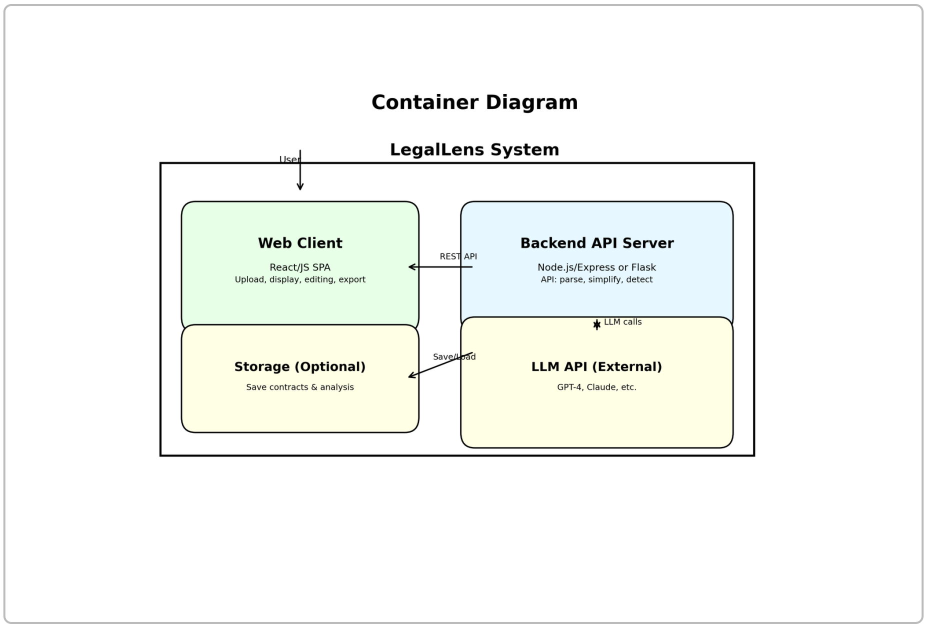


Key elements

Element	Description
User	Person uploading or pasting contracts.
LegalLens System	The application that simplifies and verifies contracts.
LLM API (External)	Large Language Model API (GPT-4/Claude) used for simplification and fraud detection.
Storage Service (optional)	Cloud storage/database for persisting contracts and analysis results.

Container Diagram

The container diagram (Level 2) zooms into the system boundary to illustrate the major containers (applications or data stores) within LegalLens and how they communicate [3]. The system is split into a client-side **web client**, a **backend API server**, an optional **storage** container, and an external **LLM API**. The arrows illustrate communication between these containers.

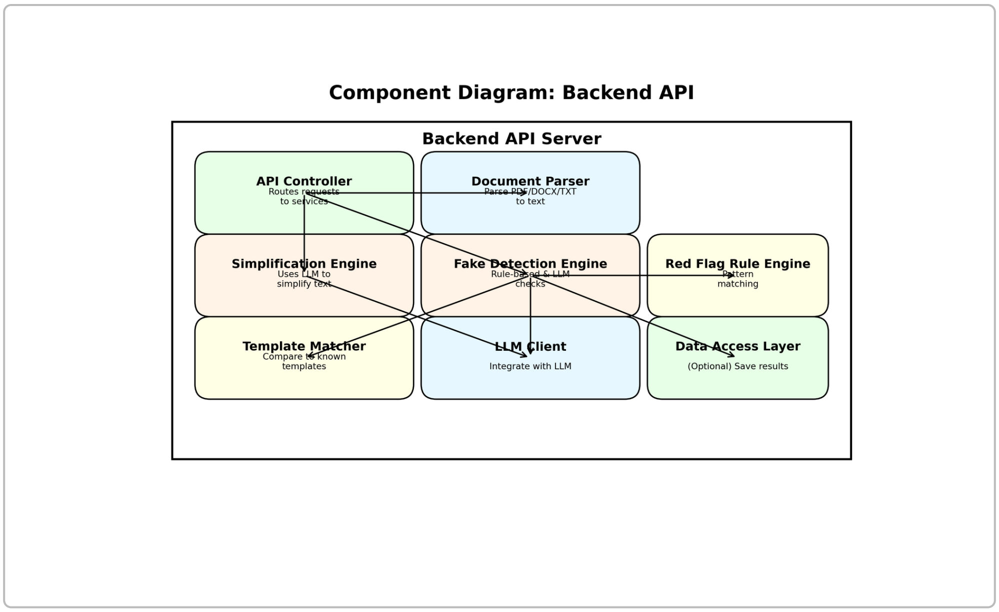


Key containers

Container	Technology & responsibilities
Web Client	React/JS single-page application running in the browser. Handles uploading/pasting contracts, displaying simplified text and red-flag reports, inline editing, and exporting results.
Backend API Server	Node.js with Express or Python/Flask. Provides REST endpoints to parse documents, simplify contracts and detect fake contracts. Delegates complex logic to the internal components described below.
Storage (optional)	Cloud storage (e.g. AWS S3) or a relational database used to persist contracts and their analysis results, if the MVP stores data.
LLM API	External service (e.g. OpenAI GPT-4, Claude). Receives contract text and returns simplified language and scam analysis.

Component Diagram (Backend)

The component diagram (Level 3) decomposes the backend API server into smaller components, illustrating their responsibilities and interactions [4]. The API server receives requests from the web client and coordinates document parsing, simplification and fraud detection. Each component encapsulates a specific responsibility, making the design maintainable.

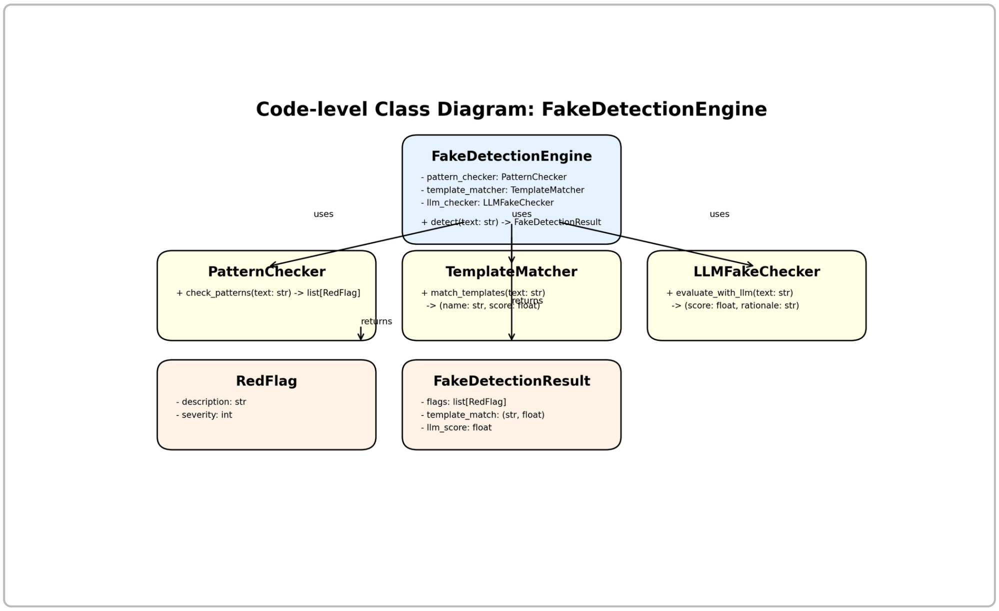


Backend components

Component	Responsibility
API Controller	Handles HTTP requests from the web client and routes them to appropriate services.
Document Parser	Converts uploaded PDF, DOCX or TXT files into plain text using libraries such as pdf.js and mammoth.
Simplification Engine	Uses the LLM client to send contract text to an LLM and returns plain-English simplifications.
Fake Detection Engine	Orchestrates rule-based and LLM-based checks to identify red flags and calculate fraud scores.
Red Flag Rule Engine	Contains a library of regex/pattern rules to detect vague parties, unusual payment terms, missing clauses, poor grammar and urgent deadlines.
Template Matcher	Compares the contract text with known templates (e.g. NDA, rental agreement) and returns a match confidence score.
LLM Client	Encapsulates the integration with external LLM APIs (OpenAI/Claude) for simplification and fraud analysis.
Data Access Layer (optional)	Persists and retrieves contracts and analysis results from storage.

Code-level Class Diagram

The optional code diagram (Level 4) zooms into the **Fake Detection Engine** component to show its internal structure using a simplified class diagram [5]. This level is usually generated on demand by tools and is most valuable for complex components. The `FakeDetectionEngine` collaborates with helper classes to perform different checks and returns a structured result.


Code-level Class Diagram: FakeDetectionEngine
FakeDetectionEngine
- pattern_checker: PatternChecker
- template_matcher: TemplateMatcher
- llm_checker: LLMFakeChecker
+ detect(text: str) -> FakeDetectionResult
uses
uses
uses
PatternChecker
+ check_patterns(text: str) -> list[RedFlag]
returns
TemplateMatcher
+ match_templates(text: str)
-> (name: str, score: float)
returns
LLMFakeChecker
+ evaluate_with_llm(text: str)
-> (score: float, rationale: str)
RedFlag
- description: str
- severity: int
FakeDetectionResult
- flags: list[RedFlag]
- template_match: (str, float)
- llm_score: float

Important classes

Class	Description
FakeDetectionEngine	Orchestrates detection by calling the pattern checker, template matcher and LLM checker. Returns a `FakeDetectionResult` containing all red flags and scores.
PatternChecker	Uses regular expressions or heuristic rules to identify problematic phrases and missing information. Returns a list of `RedFlag` objects.
TemplateMatcher	Compares text with known contract templates and returns a template name with a similarity score.
LLMFakeChecker	Sends the text to the LLM with a scam-detection prompt and parses the response to extract a fraud likelihood score and rationale.
RedFlag	Model representing an individual issue (description and severity).
FakeDetectionResult	Aggregates all red flags and scores from the different detection engines and is returned to the API controller.

Summary

The C4 model illustrates LegalLens at multiple levels of abstraction. The **system context diagram** emphasises the users and external services the system interacts with [2]. The **container diagram** shows how the web client, backend API server, optional storage and external LLM API cooperate to simplify and verify contracts [3]. The **component diagram** details the internal structure of the backend server, dividing responsibilities into logical services such as document parsing, simplification, rule-based detection and LLM integration [4]. Finally, a **code diagram** zooms into a critical component to show its classes and interactions [5]. This layered approach helps both technical and non-technical stakeholders understand how LegalLens works and provides a blueprint for implementation and future enhancements.

---

[1] Introduction | C4 model
https://c4model.com/introduction

[2] System context diagram | C4 model
https://c4model.com/diagrams/system-context

[3] Container diagram | C4 model
https://c4model.com/diagrams/container

[4] Component diagram | C4 model
https://c4model.com/diagrams/component

[5] Code diagram | C4 model
https://c4model.com/diagrams/code