2Chance Web Application

Software Dependability Project Report

Aldo Manco
Matteo Sacco
Michela Palmieri

23 January 2026

Contents

Chapter 1

Software Architectural Overview

1.1 Layered Architecture Components Breakdown

The *2Chance* platform relies on a modular layered architecture that follows the traditional Model–View–Controller (MVC) paradigm. Accordingly, system responsibilities are rigorously mapped: data presentation is managed by the View (JavaServer Pages and JavaScript); input coordination is handled by the Controller; and the Model is subdivided into Services (business logic), Data Access Objects (persistence), and Beans (data structures). This strict decoupling constitutes the existing foundation upon which the project's dependability enhancements are applied.

1.1.1 The View Layer (Presentation and Interaction)

Component: `webapp` package (e.g. `index.jsp`, `login.jsp`).
The View layer constitutes the presentational logic and is responsible for rendering the user interface (UI) and capturing user inputs. It is implemented using web standard languages, including HyperText Markup Language (HTML), Cascading Style Sheets (CSS) for styling, and JavaScript (JS) for client-side interactivity. Using JavaServer Pages (JSP), it dynamically displays data provided by the Controller typically passed via request or session attributes.

Although the View initiates the data flow by transmitting user parameters via HTTP requests, it is architecturally prohibited from executing business logic or accessing the persistence layer directly.

1.1.2 The Controller Layer (Input Coordination and Routing)

Component: `controller` package (e.g. `LoginServlet`, `RegistrazioneServlet`).
The Controller functions as the centralized entry point for server-side processing. Implemented via Java Servlets, it intercepts and manages incoming HTTP requests (specifically, `GET` and `POST`) transmitted by the client. It acts as the traffic director of the application, ensuring that user actions are routed to the appropriate business logic handlers. The Controller acts strictly as an intermediary and contains no business logic.

Its operational workflow consists of three distinct phases:

1. **Parsing:** it extracts input parameters from the request object and manages the HTTP session to preserve user state across stateless interactions.

2. **Delegation:** it invokes the appropriate methods within the Service layer, passing only the necessary arguments.

3. **Routing:** upon completion of service execution, it determines the response strategy, either forwarding the request context to a JSP view for rendering or redirecting the user to a new resource to prevent form resubmission.

1.1.3 The Service Layer (Business Logic Orchestration)

Component: `services` package (e.g. `LoginService`, `AdminService`).
The Service layer constitutes the operational core of the application, encapsulating the system's business logic. It serves as an abstraction that decouples input handling (Controller) from data storage (DAO). By isolating these concerns, the architecture centralizes business rules and makes them independent of the user interface and database technology.

Implemented as standard Java classes, Service components orchestrate complex workflows and enforce domain invariants. In particular, a Service validates inputs against application-specific rules (e.g. ensuring a password meets complexity criteria or checking stock availability) before any action is taken. To preserve architectural purity, the Service layer is prohibited from executing SQL queries directly. Instead, it coordinates persistence by invoking the Data Access layer to retrieve or persist data, ensuring that logical processing remains distinct from physical data manipulation.

1.1.4 The Data Access Layer (Persistence and DAO)

Component: `model.dao` package (e.g. `UtenteDAO`, `ProdottoDAO`).
The Data Access layer implements the Data Access Object (DAO) pattern, which abstracts and encapsulates all access to the data source. This design ensures that the rest of the application remains agnostic to the concrete database implementation, enabling potential changes in the storage mechanism without impacting higher-level logic. This layer is the only authority permitted to execute SQL commands.

Using JDBC (Java Database Connectivity) and a Tomcat connection pool for efficient resource management, DAOs manage low-level communication with the database. Their primary duties include:

1. **CRUD operations:** executing Create, Read, Update, and Delete transactions.

2. **Object–Relational Mapping (ORM):** manually transforming raw relational data (e.g. `ResultSet` rows) into high-level Java objects (Model Beans) and vice versa. This translation enables the Service layer to manipulate Java objects rather than database-specific data types.

1.1.5 The Model Beans (Data Representation)

Component: `model.beans` package (e.g. `Utente`, `Prodotto`).
Model Beans represent the fundamental functional entities of the system. Structurally, they are implemented as Plain Old Java Objects (POJOs). Their primary responsibility is data encapsulation, enabling the transport of state across the View, Controller, Service, and Data Access layers.

Each Bean mirrors a specific entity in the domain model, typically corresponding to a table in the relational database. Beans maintain state through private attributes, accessed and modified exclusively through public accessor (getter) and mutator (setter) methods.

1.2 Dynamic Behavior: The Authentication Workflow

To illustrate the dynamic interaction among the architectural components, we examine the control flow of a standard user authentication (login) procedure. This sequence shows how data propagates from the user interface to the database and back, while respecting the separation of concerns defined above:

1. **Interaction and request (View layer):** the user enters credentials in `login.jsp`. Upon submission, the browser generates an HTTP `POST` request containing the parameters (email and password) and transmits it to the server.

2. **Interception and delegation (Controller layer):** the `LoginServlet` intercepts the request and parses the body to extract input parameters. In accordance with its coordinating role, it performs no data verification; instead, it delegates the operation by invoking `login()` on `LoginService`.

3. **Orchestration and validation (Service layer):** `LoginService` receives the raw parameters and performs preliminary validation (e.g. ensuring fields are not empty). If validation succeeds, it requests data retrieval by calling `getUserByEmailPassword()` on `UtenteDAO`.

4. **Query execution (Data Access layer):** `UtenteDAO` establishes a database connection and constructs and executes a secure `SELECT` query. This is the only point in the flow where the application communicates directly with persistence storage.

5. **Object mapping (Data Access layer):** from the returned `ResultSet`, the DAO performs object–relational mapping: it instantiates a new `Utente` Bean, populates its fields with database values, and returns the object to the Service layer.

6. **State management and response (Controller layer):** the Service returns the populated `Utente` Bean to the Controller. The Controller then (i) persists state by storing the Bean into the HTTP session, effectively logging the user in, and (ii) performs navigation by sending a redirect response instructing the client to load `dashboard.jsp`, where the user's name is rendered.

Chapter 2

Testing and Coverage Analysis

In this chapter, we document the rigorous verification and validation process undertaken to enhance the dependability of the *2Chance* software platform. This project addresses the critical challenge of transforming a legacy web application into a robust, fault-tolerant system under strict resource constraints.

2.1 The Initial State: Technical Debt and Fragility

At the inception of this project, the *2Chance* platform existed as a functional but fragile legacy system. A preliminary code review revealed that the codebase lacked a defensive programming strategy: critical business-logic invariants were not validated, and runtime exceptions were frequently unhandled, leading to silent failures or undefined system states.

Furthermore, the application had zero automated test coverage. No unit or integration tests had been implemented, rendering any attempt at refactoring highly risky. This complete absence of a safety net meant that the initial reliability of the system was unquantified and presumed to be low.

The following sections detail the testing strategies selected on the basis of the system's architectural analysis. The goal is to ensure that the software is not only functional but also rigorously dependable. We employ JUnit for test execution, Mockito for component isolation, and JaCoCo for coverage analysis, using coverage results as quality-assurance metrics to assess the completeness of the intervention and to demonstrate a verified increase in system dependability.

2.2 Strategic Implementation of the Bottom-Up Testing Methodology

We adopted a bottom-up testing strategy that aligns with the dependency structure described in the *Software Architecture Overview* chapter. This methodology imposes a strict chronological constraint: any layer that depends on another must be verified only after its supporting layer has been fully validated. If testing begins at a higher layer before the underlying layers have been exercised, the effort may conclude without ever executing the supporting code. In such a case, the higher layer is typically tested against mocks, which implicitly assume that the supporting layer is correct. Consequently, a defect originating in an untested supporting layer may manifest as a failure in the higher layer, even though the higher-layer code is not at fault.

This situation confounds diagnosis, because the observed failure does not clearly indicate whether the fault lies in the layer under test or in an assumed-correct dependency. For these reasons, the testing plan follows the sequence:

$$\textit{Model Beans} \rightarrow \textit{Model DAOs} \rightarrow \textit{Services} \rightarrow \textit{Controllers}.$$

This approach ensures that we establish a foundation of reliability before introducing additional complexity.

2.2.1 Model Beans (The Foundation)

Dependency: None (independent).
Model Beans define the application data structures and their invariant constraints. The validation process must start from this layer (e.g. `Utente`, `Prodotto`) because it eliminates the most fundamental sources of error. Since every subsequent layer relies on these Beans to transport state, any defect here would propagate instability throughout the entire application.

We employ JUnit to verify that constructors and setters strictly enforce their preconditions. For example, if the `Utente` entity forbids `null` values, the corresponding unit tests must confirm that invalid inputs are correctly rejected.

2.2.2 Model DAOs (The Persistence Layer)

Dependency: Model Beans.
Once the integrity of the data structures is established, the second phase targets the Data Access Objects (DAOs). These components define the interface between the application and the persistence storage. It is imperative to verify the persistence mechanism before assessing the business logic: if the DAO layer is defective, it becomes difficult to distinguish between a logical error in the Service layer and a data retrieval failure in the DAO layer.

We use JUnit to execute Create, Read, Update, and Delete (CRUD) operations against a test database, simulated using Mockito, and we assert that the returned data matches the expected state.

2.2.3 Services (The Core Logic)

Dependency: Model DAOs and Model Beans.
The third phase verifies the Service layer (e.g. `LoginService`), which encapsulates the business logic of the application. This layer orchestrates data flow by using DAOs to retrieve information and Beans to manipulate it. This separation allows tests to focus on algorithmic correctness.

Although the DAOs have been verified in the previous phase, we strictly isolate the business logic by using Mockito to configure mock objects that simulate DAO behavior under deterministic data scenarios. Service unit tests must not interact with a live database, as doing so introduces latency and non-determinism. Since the DAOs and Beans were validated earlier, we can treat mocked interactions as reliable. Therefore, any failure detected at this stage can be attributed to the Service-layer logic.

2.2.4 Controllers (The Interface)

Dependency: Services, Model DAOs, and Model Beans.

The final phase verifies the Controllers (Servlets), which parse HTTP requests and delegate tasks to the Services. Controllers have the highest number of dependencies. If tested prematurely, a failure could originate from the Servlet itself, the Service, the DAO, or the Bean. By adhering to the bottom-up order, we ensure that the interface is built on a trusted stack, so any detected errors are attributable to HTTP handling or parameter parsing.

We again employ Mockito to mock the underlying Service layer.

2.3 The Methodological Approach: AI-Driven Unit Testing Pipeline

The integration of generative artificial intelligence has fundamentally transformed software engineering practice. AI assistants have evolved from simple auto-completion tools into proactive "pair programmers" capable of analyzing code structure, logic, and boundary conditions. In this project, we embraced this paradigm shift to maximize valid code coverage and to ensure rigorous system dependability.

To address the limitations of a small team of three developers, two of whom were new to both the system and its codebase, and a constrained timeframe (approximately 75 hours per developer), we did not manually author every test case. Instead, we used Google Antigravity, an advanced agentic coding environment in which agents have access to the entire codebase as a knowledge base and can operate directly on project files. This enabled us to develop a structured refactoring-and-testing pipeline to be executed systematically.

To achieve rigorous unit testing in isolation using JUnit 5 and Mockito, we explicitly rejected a "single-shot" prompting approach. When complex verification tasks are presented to a Large Language Model (LLM) in a single monolithic request, the resulting outputs are often characterized by diluted attention, overlooked constraints, and superficial reasoning, which can lead to "hallucinations".

Instead, we implemented a *Multi-Stage Prompting Pipeline* designed to guide the AI in generating high-quality test classes, with the objective of achieving high accuracy and the most complete unit test coverage possible. By decomposing the testing process into distinct, verifiable phases, we reduced the cognitive load on the model and enforce a disciplined reasoning workflow that supports both architectural refactoring and the construction of a comprehensive test suite. This structural decomposition ensures that the AI focuses on one verification concern at a time, resulting in deeper logical analysis and more reliable test generation.

Our Multi-Stage Prompting Pipeline is both *data-parallel* and *task-parallel*. It is data-parallel because we decompose the overall goal of testing many Java classes into independent units and generate tests for one class at a time. It is task-parallel because, for each class, we further divide test generation into three automated stages, as described below, to produce a test class that is exhaustive, robust, readable, and maintainable.

2.3.1 Automated Code Refactoring (Defensive Programming)

Reference: `src\prompts\unit_testing\01_implementation_refactoring.txt`
Input: Path of the class to refactor.

Prior to test-case generation, the codebase undergoes a rigorous refactoring step to ensure structural robustness. If business logic permits invalid states (e.g. `null` references or negative numerical values), an AI agent may generate tests that validate, rather than expose, these flaws. Consequently, our primary objective is to enforce *Design by Contract* principles. We instruct the agent to analyze the source code and implement defensive checks, ensuring that appropriate unchecked exceptions (e.g. `IllegalArgumentException`) are thrown immediately upon detecting invalid inputs. This prevents the propagation of invalid states and guarantees that the subsequent test suite is constructed on a foundation of logically sound code.

2.3.2 Automated Tests Design (Category-Partition Method)

Reference: `src\prompts\unit_testing\02_category_partition_testing.txt`
Input: Path of the class to analyze; path of the folder where Category-Partition reports will be stored.

To mitigate the bias frequently exhibited by Large Language Models where only optimal success scenarios are verified we decouple test-case design from test implementation. We employ the *Category-Partition Method* (CPM), a specification-based technique. In this phase, the agent decomposes the code into functional units, identifying *categories* (input characteristics) and *choices* (representative values). The output is a formal report of *test frames*: specific combinations of inputs that determine expected behaviors. This forces the AI to account for edge cases and boundary values before any test code is written.

2.3.3 Automated Tests Implementation

Reference: `src\prompts\unit_testing\03_unit_testing_implementation.txt`
Input: Path of the class to test; path of the Category-Partition report for the class.

The final phase translates abstract test frames into executable artifacts. The agent converts the Category-Partition report produced in Phase 2 into Java code using JUnit 5 and Mockito. Mockito is used exclusively to create fake objects for external dependencies (e.g. database connections or Data Access Objects), thereby ensuring strict unit isolation. This approach verifies each unit in a controlled environment, independent of external system stability, and produces a test suite that is well-structured and high in coverage.

Our experiments showed that providing an exhaustive chain-of-thought alone is not sufficient to ensure full test-suite correctness, nor to achieve strong readability and maintainability. Therefore, we adopted a one-shot prompting strategy augmented with a single reference example: a semi-automatically generated test class for the most complex component, `AdminService`, which includes many distinct scenarios. This example demonstrates a clear separation of test logic according to the Arrange–Act–Assert pattern, a consistent strategy for asserting expected exceptions, and a restrained use of Mockito to mock only external dependencies. In particular, mocks are used to simulate external-layer objects and to define controlled return values or behaviors, while keeping the unit under test as realistic as possible.

2.3.4 The Necessity of Human Oversight (Manual Code Review)

It is imperative to recognize that code generated by agentic AI is not always correct. Large Language Models may misinterpret requirements, make unsafe assumptions, or prioritize syntactic correctness over semantic accuracy. Consequently, a manual code-review phase is essential after each automated step to validate the integrity of the software:

1. **Post-Refactoring Review:** We verify that the AI correctly identifies invariants and that exception handling is implemented without altering the intended business logic.

2. **Post-Design Review:** We ensure that the test-frame report does not omit critical edge cases.

3. **Post-Implementation Review:** We verify that the resulting test suite is readable, maintainable, and free from hallucinations, such as tests that contradict the refactored logic (e.g. asserting that an empty password is valid when the refactoring phase explicitly forbids it).

Ultimately, the developer remains responsible for the security, compliance, and correctness of the application. The AI acts as a force multiplier, but the developer orchestrates the process and ensures that efficiency does not come at the cost of reliability.

2.4 Empirical Validation and Coverage Results

The team followed a sequential testing process in strict adherence to the bottom-up testing strategy, investing effort into each layer in the prescribed order until the time cap was reached. The `model` and `services` layers constitute the trusted architectural backbone of the application. Consequently, we performed an exhaustive verification of the foundational and logical layers: `model.beans`, `model.dao`, and `services`. The `controllers` layer currently remains untested because, from a methodological perspective, it is preferable to have a fully verified core with untested Controllers (which function as thin adaptation layers) than to maintain a verified interface that relies on unstable logic. This choice minimizes the risk of regressions where they are most costly to fix. It ensures that components with the highest reusability and dependency burden receive the highest priority, thereby maximizing system reliability within the budgeted timeframe.

The code coverage computed via JaCoCo shows that, through the adoption of Google Antigravity and our AI-driven unit testing pipeline, the project achieved an instruction coverage of 86% across the Model and Service layers. The `model.beans` layer functions as a stable data-structure foundation and achieved near-perfect instruction coverage (99%). Moreover, both the `model.dao` and `services` layers maintained a consistent instruction coverage of 85%, supporting the integrity of data persistence and business-logic rules. These results indicate that decomposing a complex verification task into granular subtasks thereby managing the context window and reducing cognitive load increased the accuracy of the AI agent.

Despite the constraints of a three-member team, with two developers new to the legacy codebase, and a limited budget of approximately 75 hours per team member, we refactored and verified a web application comprising 17 services and 16 models. This level of dependability and efficiency would have been difficult to attain using traditional manual methods. Furthermore, the pipeline exhibits potential for full automation: the sequential execution of these prompts could be orchestrated through the OpenAI API (or comparable LLM interfaces), with the additional possibility of parallelizing test generation across multiple devices, processors, and cores to further reduce time expenditure.

Table 2.1, Table 2.2 and Table 2.3 report the coverage metrics for each verified element.

Table 2.1: JaCoCo coverage metrics for the verified layers (coverage percentages).

Element	Instruction	Branch
`model.dao`	85%	74%
`services`	85%	91%
`model.beans`	99%	100%
Total	88%	86%

Table 2.2: JaCoCo coverage metrics for the verified layers (missed complexity and lines).

Element	Missed Complexity / Total	Missed Lines / Total
`model.dao`	71 / 206	88 / 664
`services`	33 / 197	82 / 537
`model.beans`	0 / 180	2 / 294
Total	104 / 583	172 / 1,495

Table 2.3: JaCoCo coverage metrics for the verified layers (missed methods and classes).

Element	Missed Methods / Total	Missed Classes / Total
`model.dao`	8 / 57	0 / 7
`services`	15 / 69	0 / 17
`model.beans`	0 / 104	0 / 9
Total	23 / 230	0 / 33

Chapter 3

Formal Specification and Verification

3.1 Formal Specification & Verification Strategy

After successfully completing the unit testing phase, we started the formal specification and verification of the code's correctness. As already discussed in unit testing, the best strategy, especially in the context of a web application, is to begin with the most atomic components, namely those that do not depend on other components and instead provide the information base used by the rest of the system.

In our application, which follows an MVC architecture, the most elementary layer is represented by the *model beans*. They model the domain entities and are used across the entire system by the other application layers (DAOs, services and controllers) as shared data structures for exchanging and managing information. Consequently, formally verifying the model beans first reduces the number of implicit assumptions: verifying more complex components would require assuming that the components they depend on are already correct, introducing a logical dependency that conflicts with the philosophy of formal verification, which aims to establish correctness through progressive and controlled steps.

A further advantage comes from the implementation nature of model beans: they typically use simple constructs and, as a rule, do not include external dependencies such as database connections, remote service calls, third-party libraries, or infrastructure components. These elements common in DAOs, services, and controllers, make formal modeling significantly more demanding because they expand the state space and increase the complexity to correctly describe the required specifications (preconditions, postconditions and invariants) in the presence of side effects or external interactions.

It is worth noting that formal verification is mainly adopted in *safety-critical* or *mission-critical* domains such as firmware, medical devices, avionics, or industrial systems, where a fault may lead to failures with severe consequences (loss of human lives, substantial economic damage or violations of regulatory requirements). In web applications, by contrast, systematic adoption of formal verification is less common because it requires a significant investment in specification writing, property validation, and debugging based on counterexamples which significantly slow software evolution, since any structural change requires coherent updates to the specifications. In such contexts, the overall cost often outweighs the expected benefits. In our case, however, the objective is educational and methodological to demonstrate that formal techniques can be applied to a selected yet meaningful portion of the system, achieving increased reliability and stronger traceability of correctness properties.

3.2 Classes Dependency Analysis

Since formal verification systematically explores the state space and requires rigorous specifications (class invariants, field constraints, method preconditions and postconditions), it is essential to reduce complexity by decomposing the work into smaller, manageable tasks. The adopted strategy is to partition the model beans into multiple groups, verifying first those with fewer dependencies and then the more interconnected ones, in order to control state-space growth and improve specification maintainability.

Our web application includes nine model beans: **Utente**, **Specifiche**, **Ordine**, **Recensione**, **Carrello**, **ProdottoCarrello**, **WishList**, **Categoria**, **Prodotto**. The first operational step is the construction of the *class diagram*, which is necessary to explicitly identify dependencies among classes and to establish a coherent verification order.

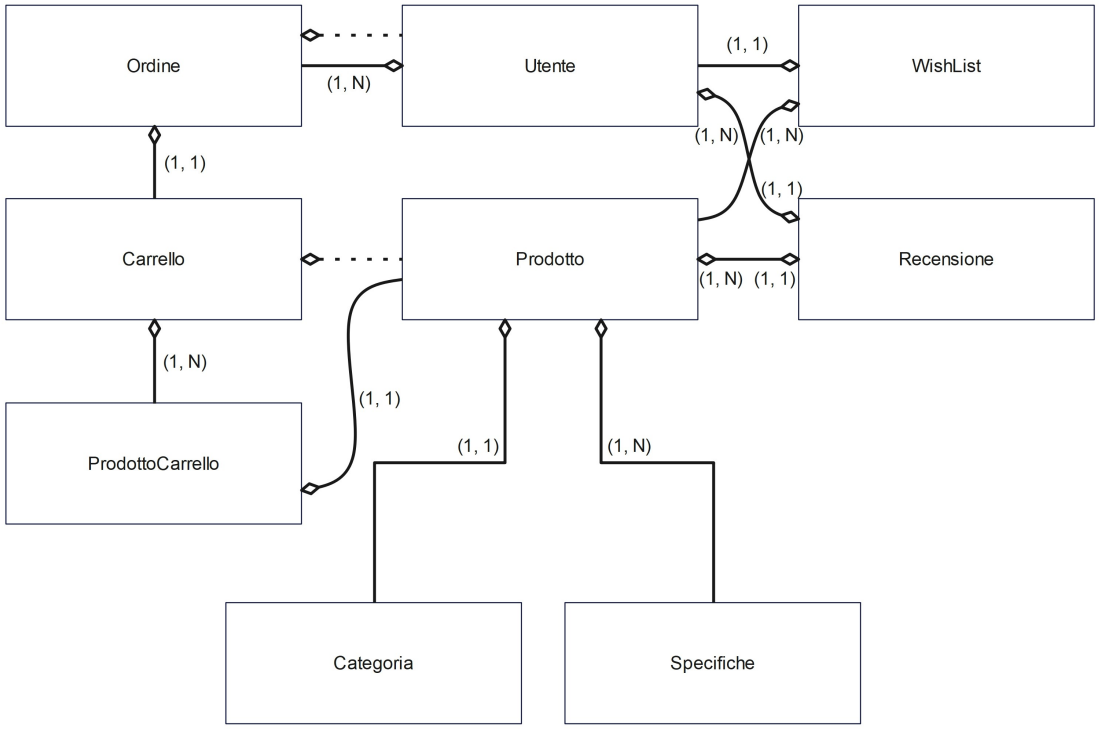

Figure 3.1: In the diagram, arrows with a diamond represent aggregation relationships (a class has, as an attribute, an instance of another class), while dashed arrows represent usage dependencies (a class uses another class within its methods or logic). This distinction is relevant because, in formal verification, both forms of dependency may affect the modeling and the set of properties that must be specified.

From the analysis of the class diagram, the following points emerge:

1. **Categoria** and **Specifiche** do not depend on other domain classes; therefore, they can be verified independently and should be addressed first.

2. There exists a **dependency cycle** involving **Prodotto**, **ProdottoCarrello**, **Carrello**, **Ordine**, **Utente**, and **Recensione**. In particular:

   - a **Prodotto** is associated with one or more **Recensioni**;
   - a **Recensione** is associated with an **Utente**;

- an **Utente** is associated with one or more **Ordini**;
- an **Ordine** includes a **Carrello**;
- a **Carrello** contains **ProdottoCarrello** items;
- a **ProdottoCarrello** refers to a **Prodotto**.

Since there is a dependency cycle, an effective verification requires specifying the involved classes jointly. Isolating a single class within the cycle would force unjustified assumptions about the others, weakening the meaning of the proof.

3. The **WishList** class depends on **Utente** and **Prodotto**; once the cyclic group has been verified, it can be verified afterwards with already established dependencies.

Based on these considerations, formal verification will be organized into the following groups, addressed in sequence:

- **First group:** Categoria
- **Second group:** Specifiche
- **Third group:** Prodotto, ProdottoCarrello, Carrello, Ordine, Utente, Recensione
- **Fourth group:** WishList

3.3 Formal Specification & Verification Plan

For each group, we adopt a standardized workflow designed to:

1. **Code simplification**: restructure each class so that its implementation relies only on basic language constructs and on methods of other classes that have already been verified;
2. **Formal specification**: define JML annotations that precisely capture the intended behaviour of the program, including:
   - **Invariants**: admissible domains and consistency constraints for class attributes;
   - **Preconditions**: admissible domains and required properties for input parameters supplied by the caller;
   - **Postconditions**: guarantees on the callee's results and on the resulting state after method execution;
   - **Assertions**: intermediate constraints on local variables and program states within method bodies;
   - **Assumptions**: properties that are taken as true when they cannot be discharged by the verifier due to unavoidable modelling or code-complexity limitations, yet are guaranteed by the surrounding system context.
3. **Formal verification and refinement**: run the verification, inspect any reported counterexamples, and iteratively refine both code guards and specifications until the considered group satisfies the required correctness properties.

As done for unit testing, these operations will be automated through a prompt pipeline manually run on the Google Antigravity environment, making the process repeatable, traceable, and less prone to manual errors. Afterwards, we manually performed an in-depth review of the AI agent's output, since it is rarely flawless. The operations are listed below in the order in which they will be executed.

3.3.1 Class Code Simplification

Reference: `src\prompts\formal_verification_and_specification\01_implementation_structure_simplification`
Input: Path of the class to refactor.

OpenJML performs formal verification by translating both Java code and its JML specifications into a mathematical/logical model typically discharged via SAT/SMT solving. For this translation to remain tractable and for specifications to be realistically writable, the code under analysis must be as simple as possible. Therefore, before writing formal specifications and running verification, the class implementation should be refactored into a verification-friendly form.

In practice, this means avoiding complex library features and opaque behaviors, limiting side effects and reducing reliance on intricate object graphs. Whenever possible, methods should use only standard-library types (e.g. `Object`, `String`, primitive wrappers) or project-defined classes that have already been specified in JML and successfully verified.

Moreover, even methods from seemingly basic classes such as `String` and `Object` should be replaced with a simple algorithm whenever possible: during our analysis, several model beans that relied heavily on `String` utilities such as `isEmpty()`, `trim()`, and `contains()` triggered errors when OpenJML attempted to translate these calls into the underlying mathematical/logical model. This design choice is the reason why formal verification of the model beans is feasible within the project time constraints: because they mainly use basic operations that can be straightforwardly replaced with simpler, verifier-friendly algorithms.

To support this step, we employ a prompt that configures a Java refactoring specialist which prepare the source code for formal verification with OpenJML. It addresses the black-box issues introduced by external methods calls (e.g. `trim()` or Java Streams) that verification engines often cannot model precisely. The task is to rewrite the class methods by replacing such library calls with explicit, primitive logic such as `for`/`while` loops and basic conditional statements, thereby making the implementation fully transparent to the underlying mathematical analysis.

A strict zero-tolerance constraint is enforced on behavioral changes: the refactored code must preserve semantic equivalence and must handle all corner cases, such as `null` values and exceptions, already present in the original implementation, such that existing unit tests are not invalidated. In this step, verifiability is explicitly prioritized over syntactic elegance or micro-optimizations.

3.3.2 Definition of Formal Specifications Using Lightweight JML

Reference: `src\prompts\formal_verification_and_specification\02_jml_annotations_definition.txt`
Input: Path of the class to annotate.

JML can express formal specifications using either a *lightweight* or a *heavyweight* style. With **Lightweight JML**, the focus is on pragmatic, broadly applicable contracts that enforce basic safety and integrity properties such as non-null references, admissible ranges, and simple consistency constraints, without explicitly characterizing every execution path.

With **Heavyweight JML**, by contrast, each relevant branch of execution is specified precisely, often separating normal and exceptional behavior and describing in detail how the

method transforms data in each scenario. Heavyweight specifications can provide strong completeness guarantees, but they are considerably more demanding to write and maintain.

In our initial attempt, we explored heavyweight specifications; however, achieving full verification coverage for all model beans with that approach would have exceeded by far the available effort budget of the team member in charge of formal specification and verification (approximately 75 hours). We therefore adopted a lightweight strategy, which enables systematic detection and correction of domain and safety violations across all model-bean classes, rather than producing an exhaustive proof for only a small subset of them.

Moreover, given that the target system is an e-commerce web application (i.e. not safety-critical), applying Design by Contract through Lightweight JML constitutes an appropriately cautious and balanced verification level.

To support this step, we employ a prompt that configures an assistant specialized in generating **Lightweight JML** specifications. The assistant receives the path of a Java file and must return the full class content augmented with JML specification blocks, **without altering the original Java code**. The prompt enforces strict constraints to avoid modeling complex operations (e.g. advanced string manipulation, streams or other constructs that are difficult for OpenJML to model reliably). Instead, it encourages simpler contracts such as non-nullness and simple range constraints, improving both runtime safety and verifiability while avoiding heavy quantification and unsustainable proof obligations.

3.3.3 Environment Setup and Verification Process

The formal verification process was conducted by a team member operating on a Windows-based system. To facilitate the use of OpenJML and its dependencies in a native Linux environment, the verification was performed using the Windows Subsystem for Linux (WSL) with an Ubuntu distribution.

To streamline the configuration of the verification environment, a shell script was developed and located at `src/scripts/openjml_ubuntu_installation.sh` to automate the installation and operational setup of OpenJML on Ubuntu systems.

Once the environment was configured, the Extended Static Checking (ESC) was executed on the entire `model/beans` package to verify the JML specifications. The following command was used to run the verification:

```
openjml -esc -progress -sourcepath /mnt/c/Users/aldom/Desktop/Projects/
   University\ of\ Salerno/2chance-dependability/src/main/java -dir /
   mnt/c/Users/aldom/Desktop/Projects/University\ of\ Salerno/2chance-
   dependability/src/main/java/model/beans
```

3.3.4 Resolution of Issues Detected by OpenJML

Reference: `src\prompts\formal_verification_and_specification\03_openjml_diagnostics_and_repair.txt`
Input: Path of the class to verify; path of the OpenJML output log.

Once the class structure has been simplified and Lightweight JML specifications have been added, we run formal verification with OpenJML. The tool translates the Java code and the

JML contracts into a logical model and checks whether any reachable state within the explored state space leads to a contradiction, meaning that the program's behavior violates the declared specifications, then it produces a counterexample. At the end, OpenJML output log is stored in a file.

To support this step, we employ a prompt that configures an AI agent to act as a **Senior Java Verification Engineer** specialized in diagnosing and fixing OpenJML reported errors. The agent receives the Java source code together with the OpenJML analysis output log, then identifies the first counterexample, maps it to the corresponding source lines, and proposes a corrective change that resolves the contradiction between code and specifications.

The key decision is to classify each OpenJML reported errors as either:

- **Real software bug**: OpenJML reports a violation that is due to an actual defect; therefore, it must be addressed by strengthening the Java code (e.g., adding defensive checks, enforcing bounds, or correcting the underlying logic);
- **False positive or an overly strict contract**: OpenJML reports a violation that is not due to an actual defect, but to modeling limitations or conservative assumptions (e.g. a field considered potentially `null` due to Java-level semantics or tool-side abstraction). In such cases, we introduce targeted assumptions or annotations that explicitly constrain the analysis so that OpenJML can rely on the intended invariants at the relevant program points, without weakening the specification globally.

During the processes of specifications definition using JML annotations and verification using OpenJML, have been introduced some specific changes to the business logic within the default constructors of **Utente**, **Recensione**, **Ordine**, and **ProdottoCarrello** to enforce class invariants from the moment of instantiation. Previously, these constructors relied on implicit default values (leaving fields null or 0), but the updated implementation explicitly initializes them to valid non-null states such as assigning empty strings, empty lists, current timestamps, or default object instances, thereby preventing potential NullPointerExceptions and ensuring strictly valid initial states for static verification.

3.4 Formal Verification Outcome

This section presents the results of the formal verification performed with `OpenJML` on the classes in the `model.beans` package. The analysis was conducted statically: `OpenJML` translated the Java implementation and its JML contracts into verification conditions (VCs) and discharged them using the SMT solver `Z3`. For each method, the log exhibits a recurring pattern: the prover is invoked, proof obligations are generated and the run terminates with outcome messages; indicating whether the solver successfully discharged the generated obligations for the inspected code or not.

Achieving the following successful outcome required approximately 30–40 refinement iterations, alternating between verification runs and targeted corrections to missing defensive guards in the implementation (e.g., null and boundary checks) and inaccurate or incomplete JML clauses, culminating in a final `OpenJML` summary report that provides the global verification counts for the successful run.

- **Valid:** 104
- **Invalid:** 0

Table 3.1: Class verification outcomes and cumulative proving time (as reported by `OpenJML`).

Class	Outcome	Time
`Carrello`	all proved	21.97 s
`ProdottoCarrello`	all proved	14.68 s
`Prodotto`	all proved	78.79 s
`Categoria`	all proved	7.10 s
`Recensione`	all proved	44.99 s
`Specifiche`	all proved	10.50 s
`Utente`	all proved	74.69 s
`Ordine`	all proved	39.47 s
`WishList`	all proved	10.53 s

- **Infeasible:** 0
- **Timeout:** 0
- **Error:** 0
- **Skipped:** 0
- **Total methods analyzed:** 104
- **Classes proved:** 9/9
- **Total duration:** 306.7 seconds

The log also reports the cumulative proving time for each class, as shown in Table 3.1.

Notably, `OpenJML` also discharged proofs for non-trivial methods (i.e., beyond plain getters/setters), including:

- `Carrello.cambiaQuantita(Prodotto,int)` (about 10.29 s);
- `Utente.hashPassword(String)` (about 12.45 s).

`OpenJML` successfully verified all classes in the `model.beans` package: every inspected method was discharged as *valid*, with no *invalid* proofs, no *infeasible* paths, no solver *timeouts*, and no tool *errors*, thereby indicating consistency between the implementation and the declared contracts. While the absence of errors provides strong evidence that the checked code satisfies the specified preconditions, postconditions, invariants, and assertions, the assurance obtained is inherently limited by the expressiveness and completeness of the specifications: a proved contract does not automatically entail full *business correctness* unless the relevant domain rules are explicitly captured in JML.

Finally, the reported proving times vary across classes and methods and typically increase with more complex control flow, richer specifications, or solver-intensive reasoning. For instance, collection-manipulating routines (e.g. cart quantity updates) and non-trivial computations (e.g. password hashing) tend to require more effort than simple setter and getter. Although proof time is not a correctness indicator, it is a useful proxy for identifying hotspots where specifications and code interactions generate more complex verification conditions.

Chapter 4

Mutation Testing

Context: PiTest

Mutation testing

Within our project, we will use mutation testing thanks to PiTest, which allows us to overcome the limitations of simple coverage (for which we use Jacoco) and identify bugs in the source code.

Tool used: pitest

Pitest was selected as the mutation testing tool, as seen in class. It allows for easy integration with Maven and supports Junit 5. The specific working version used in our project is 1.15.0.

Technical configuration and usage

Inside our pom.xml, we inserted a Maven profile named pitest that allows executing mutation testing by running the command:

```
mvn test-compile pitest:mutationCoverage -P pitest
```

The tested classes are the beans, the daos, and the services.

For the analysis, we used the standard pitest mutators:

- Condition inversion
- Return value substitution
- Arithmetic operator modification

We will find the results in the `target/pit-reports` folder.

First results

The first results highlight poor results regarding tests on daos and services.

Package Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
model.beans	9	99% (292/294)	97% (136/140)	97% (136/140)
model.dao	12	55% (433/788)	30% (103/344)	35% (103/294)
services	19	60% (339/565)	45% (96/214)	47% (96/203)
PROJECT TOTAL	51	53% (1064/1999)	42% (335/797)	52% (335/637)

The problem was resolved by increasing the tests and making them more specific.

Final results

Package Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
model.beans	9	99% (292/294)	97% (136/140)	97% (136/140)
model.dao	12	77% (606/788)	50% (173/344)	59% (173/294)
services	19	81% (456/565)	83% (177/214)	87% (177/203)
PROJECT TOTAL	51	68% (1354/1999)	61% (486/797)	76% (486/637)

Chapter 5

Performance Evaluation

5.1 Context and Goals

Performance is an important non-functional requirement: beyond functional correctness, the system should remain responsive, predictable and efficient. In a dependability-oriented project, performance is not just a user-experience concern: if critical operations degrade or time out, performance becomes an availability issue. In addition, a backend that spends too much time per request is easier to overload, which increases exposure to denial-of-service scenarios.

In this context, *performance testing* refers to the systematic evaluation of performance features of software components in order to detect performance regressions over time. In practice, it is often organized as *benchmarking*, i.e., executing an artificially generated workload against the system (or part of it) to compare versions, configurations, or environments. When the focus narrows to very small units of code (e.g., individual Java methods), the activity becomes *microbenchmarking*.

In our work, we used JMH (Java Microbenchmark Harness), an annotation-based framework designed to build, run, and analyze Java microbenchmarks. Its key value is methodological: on the JVM, just-in-time compilation and other optimizations can make naive timing measurements unstable or misleading, especially at the beginning of execution. JMH provides structured warmup and measurement phases, controlled JVM forking, and utilities (such as `Blackhole`) to reduce measurement artifacts.

The project checklist explicitly requires microbenchmarks with JMH, for this reason, the work described in this chapter focuses on *microbenchmarking* selected backend components rather than executing full end-to-end load tests. The intent is to isolate representative operations that are likely to dominate latency (data-access and service orchestration), measure them in a controlled way, and establish a baseline that can be compared across future changes.

5.2 Methodology

This chapter adopts a microbenchmarking perspective: instead of stressing the whole application end-to-end, it repeatedly executes carefully selected operations and measures their cost in isolation. JMH supports this approach by letting benchmarks declare, through annotations, how measurements should be produced (threads, forks, warmup and measurement iterations, and output units).

In our case, benchmarks are configured to report `AverageTime` (`avgt`) because the goal is to reason about latency per operation (ms/op) for DB-backed calls and for fail-fast validation paths. Other modes exist (e.g., throughput or sample time), but they were not prioritized for the baseline reported here.

5.2.1 Selection of Performance-Critical Scenarios

The starting point is the identification of "slow or expensive" use cases at the backend level. So the selection includes: database interactions (queries and inserts) and service methods that combine validation with DAO calls, often dominate request time. Therefore, the benchmark suite targets:

- DAO-layer operations (persistence access and query paths);
- service-layer operations (business logic + DAO/DB orchestration);
- connection management (connection pool acquisition/release).

A deliberate aspect of the selection is the inclusion of both *happy paths* (successful DB-backed calls) and *error/validation paths* (null inputs, invalid IDs, not-found lookups). This mix provides two kinds of evidence: (i) realistic cost for successful retrieval paths, and (ii) the overhead (or benefit) of fail-fast validation and error handling.

5.2.2 Baseline and Repeatability

Even when no optimization is applied, a first controlled run is valuable as a baseline to detect regressions. To support repeatability, the project integrates JMH into the build, uses a consistent benchmark configuration across classes, and exports raw results to a machine-readable format (JSON) to be archived as evidence and reused for reporting.

5.3 Experimental Setup

5.3.1 Project Integration and Build

JMH is integrated via Maven. Benchmark sources are located under `src/jmh/java` and included in compilation through `build-helper-maven-plugin`. The project depends on `org.openjdk.jmh:j` and `org.openjdk.jmh:jmh-generator-annprocess` (version `1.37`). The benchmark executable is packaged through `maven-assembly-plugin`, with `org.openjdk.jmh.Main` set as the main class and descriptor `src/assembly/benchmarks.xml`. The resulting artifact is the benchmark jar:

`target/benchmarks.jar`

5.3.2 Running JMH (The Workflow)

The workflow follows this approach: generate the jar, list benchmarks, run selected benchmarks, and export JSON. The same style is applied here because it is explicit, reproducible, and easy to document.

A typical run consists of:

```
mvn -DskipTests package
java -jar target/benchmarks.jar
```

For reporting and evidence collection, raw results should be exported to JSON. The measured session in this project was executed using:

```
java -jar target/benchmarks.jar ... -rf json -rff reports/performance/jmh_result.json
```

The `-rf json` option selects the JSON result format, while `-rff` specifies the output file path under the repository, so the raw output can be versioned or archived.

5.3.3 Execution Environment (Measured Run)

Benchmarks were executed under minimal system load to reduce noise and interference. The environment for the measured run is:

- OS: macOS 26.2 (Apple Silicon)
- Architecture: Apple M1 (ARM64)
- RAM: 8 GB
- Java: OpenJDK 17.0.17 (Temurin 17.0.17+10)
- System load: minimal (dedicated execution, single-threaded run)

5.3.4 Database Configuration

Several benchmarks include database calls; therefore the database setup has a direct impact on measurements. The measured run used:

- DBMS: MySQL (8.0+)
- Deployment: local loopback (`127.0.0.1:3306`)
- Database state: persistent pre-populated schema `second_chance` (shared with integration tests)
- Connection pooling: Apache Tomcat JDBC Pool (`maxActive=50`, `initialSize=5`)

With a local loopback DB and a warm connection pool, sub-millisecond scores are plausible. These values should not be generalized to remote or cold-start environments without additional runs.

5.4 Benchmark Design

5.4.1 Why JMH (and Why Not Naive Timing)

Microbenchmarking on the JVM is sensitive to warmup, JIT compilation, and aggressive optimizations. A naive `System.nanoTime()` wrapper can easily report misleading numbers (e.g., measuring cold-start behavior or having work optimized away). JMH is designed to address these pitfalls via structured warmup and measurement phases, JVM forking, and mechanisms to prevent dead code elimination.

5.4.2 Configuration Choices Implemented in Code

All benchmark classes share the same core configuration:

- `@BenchmarkMode(Mode.AverageTime)`
- `@OutputTimeUnit(TimeUnit.MILLISECONDS)`
- `@Warmup(iterations = 5, time = 1)`
- `@Measurement(iterations = 5, time = 1)`
- `@Fork(1)`

- `@State(Scope.Thread)`

This uniformity is intentional: it reduces variability caused by heterogeneous benchmark settings and makes internal comparisons easier to interpret. The choice of a single fork (`@Fork(1)`) is pragmatic (shorter total runtime); for final verification runs, increasing forks can improve robustness at the cost of longer execution time.

5.4.3 State, Setup, and Isolation

The benchmarks use `Scope.Thread` to avoid artificial contention between threads and to keep per-thread state isolated. Shared initialization is performed in `@Setup(Level.Trial)` to avoid contaminating measurements with one-off construction costs.

A concrete example is `ConnectionPoolBenchmark`, which triggers lazy initialization in the setup phase by acquiring and closing a connection before measurement starts. This ensures the benchmark focuses on the acquire/release path rather than on pool startup.

5.4.4 Blackhole and Exception-Path Benchmarking (Updated Implementation)

The updated benchmarks make an important methodological choice explicit: when intentionally measuring failure paths (null input, invalid IDs, not-found scenarios, or an unreachable DB), letting exceptions propagate may abort the benchmark execution and prevent results from being produced. For this reason, most benchmark methods catch expected exceptions (e.g., `IllegalArgumentException`, `SQLException`, and in some cases `IOException`) and consume them via `Blackhole`.

This approach keeps the harness running, makes the measured path explicit, and avoids dead code elimination because the outcome (value or error) is observed by the benchmark.

5.4.5 Input Strategy and "Fail-Fast" Paths

Input values are intentionally simple and deterministic. For instance, IDs such as `-1` represent invalid input, `99999` is used as a likely not-found ID, and `1` is used as a valid baseline entity ID on a pre-populated database. This design supports repeatability and lets the report clearly distinguish validation-only overhead from DB-backed execution.

5.5 Benchmark Suite

The repository includes eight benchmark classes under `src/jmh/java`, covering DAO, service, and infrastructure concerns:

- DAO: `OrdineDAOBenchmark`, `SpecificheDAOBenchmark`, `UtenteDAOBenchmark`, `WishListDAOBenchm` `RecensioneDAOBenchmark`
- Services: `AdminServiceBenchmark`, `LogoutServiceBenchmark`
- Infrastructure: `ConnectionPoolBenchmark`

Table 5.1 maps benchmark methods to the scenarios they exercise. This improves auditability and reduces ambiguity when interpreting results.

Table 5.1: JMH benchmark suite and exercised scenarios (updated after benchmark refactoring).

Benchmark method	Scenario
`AdminServiceBenchmark.benchmarkGetProdottoFound`	Happy path: retrieve existing product (ID 1), consumes result.
`AdminServiceBenchmark.benchmarkGetProdottoNotFound`	Error path: missing product (ID 99999), exceptions consumed.
`AdminServiceBenchmark.benchmarkInfoOrdineFound`	Happy path: retrieve existing order (ID 1), consumes result.
`AdminServiceBenchmark.benchmarkInfoOrdineNotFound`	Error path: missing order (ID 99999), exceptions consumed.
`OrdineDAOBenchmark.benchmarkGetOrdineByIdValid`	Happy path: DB fetch for order (ID 1).
`OrdineDAOBenchmark.benchmarkGetOrdineByIdInvalid`	Error path: invalid ID (`-1`), validation/exception path.
`OrdineDAOBenchmark.benchmarkGetProdottoOrdineNull`	Error path: null input, validation/exception path.
`SpecificheDAOBenchmark.benchmarkGetSpecificheValidId`	DB-dependent lookup for product ID 1 (catches `SQLException` if DB is down).
`SpecificheDAOBenchmark.benchmarkGetSpecificheInvalidId`	Invalid product ID (`-1`), exception path.
`UtenteDAOBenchmark.benchmarkGetUtenteByIdNotFound`	Attempt to find user ID `99999`, consumes returned value or `SQLException`.
`WishListDAOBenchmark.benchmarkGetWishListWithNullUtente`	Null user input, exception path.
`WishListDAOBenchmark.benchmarkGetWishListWithInvalidUtente`	Invalid user ID (`-1`), exception path.
`RecensioneDAOBenchmark.benchmarkAddRecensione`	DB insert attempt (side effect), exceptions consumed if thrown.
`LogoutServiceBenchmark.benchmarkLogoutValidSession`	Logout using a minimal `HttpSession` proxy; consumes session.
`ConnectionPoolBenchmark.benchmarkGetReleaseConnection`	Acquire/release a DB connection; requires DB availability (throws `SQLException`).

Side effects (insert benchmark). `RecensioneDAOBenchmark` measures a realistic insertion path but introduces a side effect (table growth and evolving DB state). This is useful evidence, but it must be interpreted carefully and ideally mitigated for strict comparability (see Section 5.8).

5.6 Metrics and Results Reporting

The primary metric configured in the suite is AverageTime (`avgt`), reported in milliseconds per operation (ms/op). This is a natural choice when operations are expected to be latency-dominant (e.g., DB access) and when a per-call cost is easier to interpret than aggregate throughput.

Percentile-oriented analysis (p95/p99) is not produced by the current configuration. If tail latency becomes relevant, the suite could be extended with `Mode.SampleTime` or by exporting detailed samples and analyzing distributions offline.

5.7 Results and Interpretation

Measured run scope and date. The benchmark session summarized here was executed on **2026-01-14** and intentionally focused on two components: `AdminService` (business logic + DAO/DB calls) and `OrdineDAO` (data-access layer). This provides a clean first baseline contrasting happy paths with fail-fast/error paths.

Traceability note. The following Table 5.2 reports the measured Score (AverageTime, ms/op) for the selected scope; additional JMH statistical fields (e.g., Cnt and Error) are available in the JSON report at reports/performance/jmh_result.json.

Table 5.2: Measured JMH results (AverageTime, ms/op) for the selected scope.

Benchmark	Scenario	Score
`AdminServiceBenchmark.benchmarkGetProdottoFound`	Happy path (ID 1)	0.079
`AdminServiceBenchmark.benchmarkGetProdottoNotFound`	Error path (missing)	0.139
`AdminServiceBenchmark.benchmarkInfoOrdineFound`	Happy path (ID 1)	0.216
`AdminServiceBenchmark.benchmarkInfoOrdineNotFound`	Error path (missing)	0.085
`OrdineDAOBenchmark.benchmarkGetOrdineByIdValid`	Happy path (ID 1)	0.104
`OrdineDAOBenchmark.benchmarkGetOrdineByIdInvalid`	Error path (ID -1)	0.001
`OrdineDAOBenchmark.benchmarkGetProdottoOrdineNull`	Error path (null)	0.001

Interpretation. The results align with a common backend intuition: fail-fast paths (invalid IDs or null inputs) are effectively negligible compared to DB-backed execution. In this run, the gap is roughly two orders of magnitude (0.001 vs 0.104 ms/op), indicating that input validation prevents unnecessary DB round-trips and avoids consuming pool resources—an effect that can matter under sustained load.

For `AdminService`, found vs. not-found is not trivially predictable: not-found can be slower due to exception handling and additional checks, or faster due to early exit. Here, `infoOrdineNotFound` is faster than the found path, while `getProdottoNotFound` is slower than the found path. This kind of contrast is useful in practice: it tells us where error paths might be disproportionately expensive and where the implementation already fails quickly.

Finally, scores below 1 ms/op are consistent with the measured setup (MySQL on loopback, pre-populated database, and a warm connection pool). These results should be treated as a local baseline rather than a general performance claim for remote deployments.

5.8 Threats to Validity

JVM and OS noise. Microbenchmarks on the JVM are affected by OS scheduling jitter, CPU frequency scaling and thermal behavior, background activity, and garbage collection. Running on a quiet machine reduces variance, but it does not eliminate it. Using additional forks and repeating runs can improve confidence.

Database state and caching. DB-dependent benchmarks are highly sensitive to database state: warm vs cold caches, buffer pool contents, index locality, and table size all affect latency. Results obtained with a persistent pre-populated database (shared with tests) should be interpreted accordingly. If strict comparability is needed, DB state should be reset or standardized between runs (e.g., snapshot/restore).

Side effects (insert benchmark). Benchmarks that mutate the database (e.g., inserting reviews) can drift over time as tables grow and indexes evolve. A clean mitigation is to reset the database state between runs, or to execute inserts inside transactions that are rolled back. Where such mitigation is not applied, results should be treated as indicative rather than strictly comparable.

Exception cost and semantic mismatch. Exception-path benchmarks can be disproportionately expensive because exception creation and handling may dominate time. These measurements should be interpreted as robustness/guard-logic overhead, not as a proxy for the latency of successful DB operations.

CI execution. Running performance benchmarks in standard CI runners often produces noisy and misleading results due to shared resources. A more defensible approach is to run JMH on-demand or on a controlled baseline machine, always recording environment, command line, and raw evidence.

5.9 Conclusions and Recommendations

The project implements a coherent JMH suite integrated into the Maven build and targeted at plausible backend hotspots (DAO operations, service orchestration, and connection pooling). The benchmark design reflects the key methodological points: structured warmup/measurement, JVM forking, thread-scoped state, and explicit handling of values/exceptions to reduce optimization artifacts.

The measured baseline (2026-01-14) confirms a practical architectural benefit: fail-fast validation avoids spending DB round-trip time on invalid input, while the service-level benchmarks provide early insight into how success and failure paths differ in cost. The results are strong evidence for the value of defensive validation and for tracking regressions over time.

Chapter 6

Containerization and Orchestration Readiness

Context: Docker & DockerHub

A containerization strategy has been prepared for our project to distribute the project more easily, ensuring portability, dependency isolation, and consistency between development environments.

Containerization Strategy

A Dockerfile was created that builds our application in multiple steps:

- **War file build**: through the command `mvn clean package -Dmaven.test.skip=true`, the war file is created on a `maven:3.9-eclipse-temurin-19` image
- **Application**: Copying of the war file inside the `tomcat:9.0` image
- **Network**: port 8080 is exposed to access the application

Docker compose

Since our architecture includes a database that is currently not hosted online, it was necessary to create a docker compose that initializes both the application image created previously and the database.

We will therefore have two services:

- **webapp**: the application created previously
- **mysql-db**: the mysql database which is initialized on a `mysql:8.0` image. Furthermore, the db is initialized with data from the dump `db_import.sql`

CI/CD

Containerization is central to our automation pipeline managed with GitHub Actions. In fact, upon every push, a github action will be executed that will rebuild the image and execute all tests and necessary operations on the running docker image.

Indeed, the last operation of our action is the push of our image to docker hub via the following step:

```
- name: Build and push Docker image
    uses: docker/build-push-action@v5
    with:
      context: .
      push: true
      tags: ${{ secrets.DOCKER_USERNAME }}/2chance-dependability:latest
```

Chapter 7

Security Assurance and Vulnerability Assessment

7.1 Context and Objectives

In a dependability-oriented project, security assurance is treated as part of operational reliability: defects that expose secrets, pull vulnerable libraries, or leak runtime details can quickly become availability incidents (e.g., compromised credentials, exploit-driven downtime, or denial-of-service amplification). The goal of this chapter is therefore not to claim "perfect security", but to document an hardening process applied to the *2Chance* codebase.

The analysis focused on three complementary assessment dimensions:

- **Secret detection** at repository level (prevent accidental credential exposure);
- **Software Composition Analysis (SCA)** for vulnerable dependencies (supply-chain risk);
- **Static Application Security Testing (SAST)** for risky code patterns (defensive coding at entry points).

The work was conducted iteratively following the flow: *Scan through CI pipeline in GitHub* → *Fix the issues* → *Push on the repository* → *Re-Iterate*. All outputs, notes, and before/after artifacts are stored under `reports/evidence/security/`.

7.2 Background on Security Tools

In this project, we employ a structured security approach using three tools, each addressing a specific dimension of software security. This section provides a high-level overview of these tools before detailing their specific integration in our pipeline.

7.2.1 GitGuardian (Secret Detection)

GitGuardian is a platform designed to detect sensitive information (such as API keys, database credentials, and certificates) that may be accidentally committed to source control. By scanning both the git history and incoming commits, it prevents "secrets sprawl," ensuring that hardcoded credentials do not bypass access controls or leak to unauthorized parties.

7.2.2 Snyk (Software Composition Analysis)

Snyk is a security tool that focuses on Software Composition Analysis (SCA). It scans open-source dependencies (in this case, Maven libraries defined in `pom.xml`) against a database of known vulnerabilities (CVEs). It helps manage supply-chain risk by identifying outdated or compromised libraries and suggesting upgrade paths or patches.

7.2.3 SonarQube Cloud (Static Analysis)

SonarQube Cloud is a cloud-based Static Application Security Testing (SAST) and code quality service. It analyzes source code for bugs, vulnerabilities, and code smells without executing the program. In the context of this project, it is particularly used to enforce secure coding standards in the Java implementation, identifying patterns that could lead to reliability or security issues (e.g., improper exception handling).

7.3 Process, Toolchain, and Evidence Trail

Security analysis is integrated into the GitHub Actions workflow (`.github/workflows/main.yml`) and relies on the following toolchain:

- **GitGuardian**: Executed immediately after repository checkout to detect secrets or credentials in the codebase. It scans commits (push and pull requests) and prevents the merge of sensitive data.
- **Snyk**: Runs as a dedicated step to identify vulnerabilities in Maven dependencies. The pipeline is configured to fail the build if *High* or *Critical* severity issues are detected (`-severity-threshold=high`).
- **SonarQube Cloud**: Integrated into the Maven build lifecycle. It performs static code analysis to detect bugs, code smells, and security hotspots.

Evidence is organized per tool:

- `reports/evidence/security/gitguardian/` (scan outputs, fixes, walkthrough);
- `reports/evidence/security/snyk/` (baseline scan, post-fix scan, dependency changes);
- `reports/evidence/security/sonar/` (CSV/JSON issue exports and remediation walkthrough).

7.3.1 Before/After Summary

Table 7.1 summarizes the before/after state as captured by the three tools. Numbers are taken from the baseline reports and the final verification runs stored in the evidence folders.

Table 7.1: Security assessment summary (before/after) based on stored evidence.

Tool	Before	After	Outcome
GitGuardian	6 occurrences (Generic Password)	0	Resolved
Snyk	14 issues total	0	Resolved
SonarQube Cloud	96 issues	0	Resolved

7.4 Security Mechanisms in CI/CD

The repository includes a comprehensive GitHub Actions workflow (`.github/workflows/main.yml`) that enforces security checks before deployment. The pipeline defines a job `build-secure-and-deploy` that sequentially executes:

1. **Secret Scanning**: via GitGuardian action;
2. **SCA**: via Snyk CLI;
3. **Build & SAST**: Maven build with SonarQube analysis;
4. **Docker Build**: pushes the image only if all previous steps succeed.

These checks act as automated "quality gates", ensuring that code violating security policies breaks the build and prevents deployment.

7.5 GitGuardian: Secret Detection and Remediation

7.5.1 Why Secret Scanning

Secrets committed to a repository have a disproportionate blast radius: a single leaked password or token may enable unauthorized access and bypass many other controls. Secret scanning aims to prevent this class of incident early, before credentials are reused or propagated across forks and clones.

7.5.2 Baseline Findings

The baseline GitGuardian repository scan detected **two instances** of the *Generic Password* class, for a total of **six occurrences** distributed across two files:

- `src/prompts/03\_unit\_testing\_implementation.txt` (documentation content);
- `src/test/java/services/RegistrazioneServiceTest.java` (unit tests).

Raw evidence is stored in:

- `reports/evidence/security/gitguardian/ggshield\_repo\_scan.txt` (baseline);
- `reports/evidence/security/gitguardian/ggshield\_repo\_scan\_after\_cleanup.txt` (post-fix).

7.5.3 Remediation and Rationale

A conservative remediation strategy was used, instead of simply "marking as safe" in the dashboard, we applied a configuration-as-code approach to ensure reproducibility:

Configuration Changes. A `.gitguardian.yaml` file was introduced to manage exclusions directly in the repository. Key changes include:

- **Path Exclusions**: `secret.ignored_paths` was configured to exclude `src/prompts/**` and unit tests (`src/test/**`), distinguishing between real secrets and test data.
- **Pattern Matching**: `secret.ignored_matches` was updated to ignore known test credentials (e.g., `Pass123!`, `validPassword`), reducing noise from legacy test mocks.

This configuration reduced the signal-to-noise ratio, ensuring future scans focus on genuine leaks. The applied changes and rationale are documented in: `reports/evidence/security/gitguardian/security\_analysis\_gitguardian.md` and `reports/evidence/security/gitguardian/gitguardian\_fixes.md`.

7.5.4 Verification

After cleanup, GitGuardian reported **0** detected secrets in the repository scan, as recorded in `ggshield\_repo\_scan\_after\_cleanup.txt`.

7.6 Snyk: Dependency (SCA) Vulnerability Assessment

7.6.1 Why SCA

Modern applications inherit a significant portion of their attack surface from third-party libraries. Software Composition Analysis focuses on known vulnerabilities (e.g., CVEs) in direct and transitive dependencies. The typical remediation is to upgrade to a patched version or migrate to an actively maintained alternative (maintained artifacts coordinate).

7.6.2 Baseline Findings

The baseline scan output (`reports/evidence/security/snyk/snyk\_test.txt`) reports **11 vulnerable paths** at first pass, which were consolidated in the analysis notes as **14 issues total** across remediation phases. The affected components clustered around:

- `commons-io:commons-io` (resource exhaustion);
- `org.json:json` (DoS-related issues);
- MySQL connector and its transitive chain;
- JSTL (`javax.servlet:jstl`) (XXE and legacy namespace).

7.6.3 Remediation

Two kinds of fixes were applied:

- **Version upgrades** for maintained artifacts (e.g., `commons-io` from `2.10.0` to `2.14.0`, `org.json:json` from `20210307` to `20231013`);
- **Artifact migration** where the old coordinate was deprecated or unpatched.

Critical Migrations. Specific attention was paid to two critical findings:

1. **MySQL Connector**: Migrated from `mysql:mysql-connector-java` to `com.mysql:mysql-connect` (version `9.5.0`) to address Buffer Overflow vulnerabilities in the legacy 8.x driver.
2. **JSTL**: Replaced the vulnerable `javax.servlet:jstl` with `org.apache.taglibs:taglibs-standar` (`1.2.5`) to resolve XXE Injection risks while maintaining compatibility with the Servlet API.

The exact changes and their rationale are documented in: `reports/evidence/security/snyk/security\_analysis\_snyk.md` and `reports/evidence/security/snyk/snyk\_fixes.md`.

7.6.4 Verification

A final scan (`reports/evidence/security/snyk/snyk\_test\_after\_cleanup.txt`) reports **0 issues**. In addition, a full build and test run was executed to reduce the risk of dependency-related regressions (see the Snyk fixes notes; test suite result: 533 tests passed).

7.7 SonarQube Cloud: Static Security Analysis on the Controller Layer

7.7.1 Why SAST for Servlets

Static analysis is particularly useful at application entry points: in servlet-based architectures, controller methods (`doGet`, `doPost`, lifecycle hooks) are a common source of uncontrolled exception propagation and accidental information leakage. Even when the underlying bug is "only" an exception, the security impact is often practical: stack traces and internal error details can reveal system structure and enable targeted attacks.

7.7.2 Baseline Findings

SonarQube Cloud reported **96 issues** mapped to the rule `java:S1989` ("Exceptions should not be thrown by servlet methods"). Raw exports are stored in:

- `reports/evidence/security/sonar/sonar\_security\_issues.csv` (initial);
- `reports/evidence/security/sonar/sonar\_security\_issues\_2.csv` (mid-remediation);
- `reports/evidence/security/sonar/sonar\_security\_issues\_final.csv` (final).

7.7.3 Remediation Pattern and Motivation

The remediation targeted **17 Servlet controllers** (including `LoginServlet`, `AdminServlet`, and `RegistrazioneServlet`). The fix involved a systematic refactoring to prevent unchecked exceptions from reaching the container's default error page.

Implementation Details.

- **Safe Error Handling**: A new utility `src/main/java/utils/ResponseUtils.java` was introduced to centralize error responses. It uses `ResponseUtils.sendError(response, status, message)` to handle IOExceptions recursively, ensuring that a failure in error reporting does not crash the servlet.
- **Standardized Wrapping**: All `doGet`, `doPost`, and `init` methods were wrapped in 'try-catch' blocks. Caught exceptions are logged and converted to generic 500 error responses, masking stack traces from the client.

The full walkthrough and file list are documented in: `reports/evidence/security/sonar/walkthrough\_sonar.md` and `reports/evidence/security/sonar/sonar\_fixes.md`.

7.7.4 Verification

The final Sonar exports report **0 issues**. A Maven build and full test suite execution were performed after the refactor (533/533 tests passed), providing additional confidence that changes were behavior-preserving.

7.8 Web Application Shows No Vulnerabilities

Static tools and dependency scanners reduce risk, but they are not sufficient to claim that the *running* web application "shows no vulnerabilities". This section should be addressed with a final, system-level verification step (dynamic scanning and/or targeted manual checks) against a deployed instance of the application.

Evidence-based claim (scope). Based on the final re-scans, the security campaign reports no remaining findings in its scope: GitGuardian detects 0 secrets in the repository, Snyk reports 0 known vulnerable dependency issues, and SonarQube Cloud reports 0 occurrences of rule `java:S1989`. Therefore, the application shows no detected vulnerabilities.

Completeness note. For completeness, a lightweight dynamic verification (DAST) against a running deployment can be performed (e.g., baseline scanning of the exposed endpoints) and archived as additional evidence.

7.9 Threats to Validity

False positives and rule interpretation. All three tools may produce false positives or findings that require contextual interpretation. GitGuardian findings on "generic passwords" are particularly sensitive to documentation and tests; Snyk reports may change as advisories evolve; Sonar rules may flag patterns that are not harmful in specific contexts. For this reason, human supervision is an explicit part of the documented process.

Environment and reproducibility. Tool results depend on execution context: local configurations, authentication state for cloud dashboards, and repository history. To mitigate this, baseline and post-fix outputs were archived in the repository. Still, exact outputs may differ across time due to updated vulnerability databases (Snyk) or updated analyzer rules (Sonar).

Dependency upgrades and behavioral drift. Upgrading dependencies improves security but may introduce subtle compatibility changes. The project mitigated this risk by rebuilding and running the full test suite after remediation. Residual behavioral drift is still possible and should be monitored in future changes.

Coverage gaps without dynamic testing. Without the final section verification 7.8, the assessment remains incomplete at the runtime layer. Static and SCA results should be interpreted as "reduced risk" rather than "no vulnerabilities" for the deployed application.

7.10 Conclusions and Next Steps

The security assurance work followed a concrete scan–fix–verify loop and produced repository-stored evidence for each tool. The project removed: (i) secret-like patterns from documentation/tests and constrained false positives via `.gitguardian.yaml`, (ii) dependency vulnerabilities through upgrades and targeted artifact migrations, and (iii) risky servlet exception propagation patterns by introducing consistent controller-side error handling and a shared `ResponseUtils` utility.

Chapter 8

Build and Deployment Pipeline

Context: Security in CI/CD

Build and deployment pipeline

To automate all testing, deployment and build operations a github action has been set up that is executed at every push.

Overview

Our 'main.yml' pipeline ensures that every change is ready and safe for production.

- **trigger**: The workflow activates at every push on the main branch (main)
- **environment**: The entire workflow runs on a virtual machine based on 'ubuntu-latest'

flow:

In our pipeline multiple steps are executed, in order:

- Code checkout
- Security scan with GitGuardian
- Dependency vulnerability analysis with Snyk
- Java 19 environment setup and dependency caching
- MySQL database initialization with data import
- Build, unit tests and mutation coverage, static analysis with SonarQube
- Upload of coverage results to Codecov
- Login to Docker Hub, image build and push

Steps in detail

In this paragraph we analyze some fundamental steps of our github action.

Security scan with GitGuardian:

thanks to sending the current commit data to GitGuardian we initialize a code scan on the latest changes. This will allow verifying the presence of exposed passwords or secrets, blocking pipeline execution in case vulnerabilities are identified.

Dependency vulnerability analysis with Snyk:

thanks to the use of the tool 'snyk' we identify security weaknesses classified according to CWE through dependency analysis.

Mysql:

inside the github action a test database with test data is instantiated. This is necessary because otherwise most tests would fail due to missing connection to the db. This solution allows running tests on a totally isolated, reproducible and modifiable environment.

Build, Unit Testing and Mutation Testing:

Through the command 'mvn -B clean verify -P pitest' we are able to build, launch unit tests and mutation tests on our application. Through the maven profile 'pitest' mutants are introduced in packages model.beans, model.dao and services.

The test results are subsequently uploaded to the 'Codecov' platform which will allow us to visualize results and will manage the failure or success of the push thanks to the key '`fail_ci_if_error`: true'. Furthermore a static code analysis is performed via 'SonarQube' with the flag '`-Dsonar.qualitygate.wait`=true' which allows blocking the pipeline in case of quality weaknesses.

Additional security:

Inside our github action we will not find credentials but references to secrets saved in the github repo, such as for example

```
- name: Log in to Docker Hub
     uses: docker/login-action@v3
     with:
       username: ${{ secrets.DOCKER_USERNAME }}
       password: ${{ secrets.DOCKER_PASSWORD }}
```

Docker hub:

Finally if all the the steps of the pipeline have been executed successfully the build and push of the docker image on docker hub is performed.