CHALMERS
UNIVERSITY OF TECHNOLOGY

AI-Powered Slack Chatbot for Design System Compliance & Collaboration

Design and Evaluation of an LLM-Based Workflow for Detecting Inconsistencies Between Figma Design Components and Frontend Implementations

Master's thesis in Complex Adaptive Systems

CARL KYLEBÄCK WENNERLÖF

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

AI-Powered Slack Chatbot for Design System Compliance & Collaboration

Design and Evaluation of an LLM-Based Workflow for Detecting Inconsistencies Between Figma Design Components and Frontend Implementations

CARL KYLEBÄCK WENNERLÖF

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2026

AI-Powered Slack Chatbot for Design System Compliance & Collaboration

CARL KYLEBÄCK WENNERLÖF

Supervisor: Juntima Nawilaijaroen, Ericsson
Examiner: Andreas Ekström, Chalmers University of Technology

Master's Thesis 2026
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Maintaining consistency between design specifications and frontend implementations is a recurring challenge in large-scale software development. In industrial design systems, user interface components are often defined by designers in tools such as Figma and implemented separately by developers in frontend codebases. Because these two representations are created, maintained, and updated in separate environments, inconsistencies can appear over time. This can lead to design-code drift, where visual and structural properties such as colors, spacing, typography, layout, and component variants no longer align.

This thesis investigates how an AI-based system can be designed to detect and explain inconsistencies between Figma-based design components and their corresponding frontend implementations. A proof-of-concept system was developed in an industrial context at Ericsson. The system retrieves design data from Figma and frontend implementation data from a GitLab-hosted codebase, filters and transforms both sources into structured JSON representations, matches corresponding components using a hybrid fuzzy and LLM-based matching approach, and performs compliance checking using a controlled LLM-based comparison pipeline. The results are delivered through a Slack chatbot interface to support integration with existing developer and designer workflows.

The system was evaluated using industrial design-system components and controlled test cases with known inconsistencies. The evaluation examined component retrieval, component matching, inconsistency detection, prompt strategies, model selection, data reduction, token usage, and practical workflow behavior. The strongest configuration, using filtered JSON, GPT-5.1, and multi-pass prompting, achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510 for inconsistency detection. The matching pipeline produced matches for 109 of 189 components, while the wrong-match correction mechanism identified and rejected 10 initially accepted incorrect matches.

The results show that LLM-based design-code compliance checking can be effective when used as part of a controlled pipeline with deterministic preprocessing, structured representations, schema-constrained outputs, and task-specific prompting. However, the results also show that automatic component matching, scalability, latency, and human trust remain important challenges. The thesis shows that AI-based compliance checking is best understood as a support tool for developers and designers rather than a fully automatic source of truth.

Keywords: design systems, large language models, LLM agents, design-code compliance, frontend development, Figma, GitLab, Slack chatbot, LangChain, LangGraph, software engineering, AI-assisted development.

Acknowledgements

I would like to thank Ericsson and my Supervisor Juntima Nawilaijaroen for providing the opportunity to conduct this thesis in an industrial environment and for giving access to the systems, workflows, and technical support needed throughout the project. I also would like to thank my supervisors and colleagues for their feedback, discussions, and guidance during the development and evaluation of the system.
I would also like to thank my examiner, Andreas Ekström, for reviewing the thesis work and providing academic guidance.

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AAD Azure Active Directory
AI Artificial Intelligence
API Application Programming Interface
CI/CD Continuous Integration / Continuous Deployment
CSS Cascading Style Sheets
DOM Document Object Model
DSR Design Science Research
F1 F1-score
HTML HyperText Markup Language
HTTP HyperText Transfer Protocol
JSON JavaScript Object Notation
LLM Large Language Model
MVP Minimum Viable Product
NLP Natural Language Processing
OAuth Open Authorization
REST Representational State Transfer
ReAct Reasoning and Acting
SDK Software Development Kit
UI User Interface
UX User Experience

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Modern software development often relies on *design systems*. These systems are structured collections of reusable UI components, design tokens, and guidelines to create visual consistency, usability, and scalability between products [1]. Design systems help organizations coordinate work between designers and developers by defining standardized components and shared styling rules. In large enterprises, these systems can contain hundreds of components in multiple teams and product lines.

Designers usually create components using design tools such as Figma, while developers implement the corresponding functionality in frontend frameworks such as React or Ember [2]. Even though both represent the same components, they are implemented in different ways, structured design metadata in Figma and executable source code in the frontend codebase. The design of these components must be the same to maintain visual consistency and user experience.

Discrepancies often appear between design specifications and implementations. Small differences in spacing, typography, color values, or layout logic can degrade the user experience. As systems evolve and components are modified separately by designers and developers, inconsistencies may increase, which is usually called architectural or design drift [3]. Manual compliance verification is time consuming and scales poorly in large component libraries, especially when updates occur in both design files and code repositories.

The challenge is harder in industrial settings where design systems are continuously worked on [4]. Ensuring compliance between design artifacts and frontend implementations becomes an ongoing issue instead of a one time validation task. Existing approaches to compliance checking in software engineering are mostly semi-automated and often depend on manual review, while automated techniques are often rule based and limited in scope.

Recent advances in artificial intelligence (AI), and in particular large language models (LLMs), have shown strong capabilities in reasoning about source code and structured representations of data [5]. LLMs can interpret complex code structures, understand semantic relationships between components, and generate structured analysis of discrepancies. These capabilities shows the potential for automated systems that assist in detecting and explaining inconsistencies between design specifications and frontend implementations.

However, there has not been much work that has explored the integration of LLM-

based reasoning into structured design–implementation compliance workflows for industrial design systems. There is a lack of research on hybrid approaches that combine deterministic preprocessing with semantic analysis to get robustness and reproducibility. This thesis addresses this gap by developing and evaluating an automated architecture for comparing Figma-based design components with their corresponding frontend implementations.

1.2 Problem Statement

Although design systems are intended to ensure consistency between design specifications and implemented user interfaces, keeping this consistency between design and frontend code is still a challenge. At Ericsson, UX designers define component behavior and visual properties in Figma, while developers implement the corresponding components in a React or Ember-based frontend codebase. Because these artifacts are worked on independently by different people, discrepancies between design specifications and implemented components often appear over time.
Identifying these discrepancies requires manual inspection. Designers and developers must manually compare each Figma components with their frontend implementations to verify compliance with the design system. This process is time consuming, error prone, and difficult to scale when the number of components and design variants grows. Manual verification often occurs late in the development cycle, allowing inconsistencies to exist for extended periods or missed entirely.
Existing development tools provide limited support for verifying consistency between design artifacts and implemented components. Static analysis tools operate on code but typically lack access to design specifications, while design tools focus on visual aspects rather than automated verification of code implementations. As a result, there is currently no automated workflow at Ericsson that systematically compares Figma components with their corresponding frontend implementations.

1.2.1 Research Questions

The overall research question guiding this thesis is:

1. **RQ1:** How can an AI-based system be created and used to detect and explain inconsistencies between UX design components and their corresponding frontend implementations?

To address this overarching question, the thesis investigates the following sub-research questions:

1. **RQ1.1:** How can design components from Figma and frontend components from a GitLab-hosted codebase be reliably retrieved and represented in a form suitable for AI-based comparison?
2. **RQ1.2:** How can design components from Figma be automatically matched to their corresponding frontend implementations when naming conventions differ between the two sources?
3. **RQ1.3:** How can a comparison pipeline be designed that combines structured design data and frontend code representations to support LLM-based detection of design–code inconsistencies?

4. **RQ1.4:** How can a Slack-based interface be designed to support querying, reporting, and notification of design-code inconsistencies in an helpful and clear way?

1.2.2 Scope and Limitations

This thesis is scoped to the context of Ericsson's internal design system and the specific tools and technologies used within that environment. The design source is limited to components defined in Figma, and the frontend implementation is based on React and Ember using styled-components for styling.
The solution is developed as a proof of concept for internal use and is not intended to be a general-purpose design compliance tool. It assumes a specific project structure, a shared theme file for design tokens, and access to both the Figma API and a GitLab-hosted frontend repository. While the underlying approach could be adapted to other design tools, frontend frameworks, or component architectures, that is outside the scope of this work.

1.3 Report Outline

Chapter 2 presents the theoretical background and related work on design systems, design-code compliance, large language models, agents, and component matching. Chapter 3 describes the research methodology, data collection, evaluation approach, and ethical considerations. Chapter 4 presents the design and implementation of the AI-powered Slack chatbot and its supporting data, matching, and compliance-checking pipelines. Chapter 5 presents the results from the component data pipeline, component matching, compliance checking, system performance, and user feedback. Chapter 6 discusses the results in relation to the research questions, design decisions, limitations, and future work. Chapter 7 concludes the thesis.

2
Theory and Related Work

2.1 Design Systems

Design systems have become a central concept in modern software development, particularly in organizations that maintain multiple digital products or large user interfaces [1]. A design system is defined as a structured collection of reusable user interface components, design tokens, guidelines, and documentation that together ensure visual consistency across products. By standardizing visual elements such as buttons, forms, navigation patterns, and typography, design systems help teams to build interfaces that are consistent, maintainable, and scalable.

Designers define component behavior, layout rules, and visual properties using design tools, while developers implement these components in frontend frameworks. When they both rely on the same design system definitions, teams can reuse components instead of repeatedly designing and implementing similar interface elements. This reduces duplication of effort and supports more efficient development workflows.

Large organizations often have extensive design systems that evolve over time as new features, products, and branding requirements occur. As a result, design systems are continuously updated, extended, and refactored. Although this evolution enables flexibility and reuse, it also introduces challenges in maintaining consistency between design specifications and the corresponding frontend implementations. Making sure that implemented components stay compliant with the design system becomes an ongoing engineering and collaboration task.

2.1.1 Figma as a Design Tool

Figma is often used in modern UX and product design workflows as a collaborative interface design platform [7]. It allows designers to define visual layouts, interaction patterns, and reusable components in a shared design environment. In many organizations, Figma design files serve as the primary specification of user interface components before they are implemented in code.

A central concept in Figma is the *component*. Components represent reusable UI elements such as buttons, input fields, or navigation elements. Figma supports grouping related components into *component sets* that allow designers to define multiple *variants* of the same component. Variants typically represent different states or configurations of a component, for example color variations, theme variations (e.g., light and dark), or interaction states such as hover or disabled.

Figma also provides layout mechanisms such as *auto-layout* that helps designers to define alignment, spacing, and responsive behavior between elements. These layout properties capture important structural information about how components should behave when rendered in an interface. This makes design artifacts created in Figma contain structured information about visual properties, layout rules, and component relationships.

In addition to its graphical interface, Figma exposes a REST API that allows external systems to programmatically access design data. The API provides access to the document tree, component metadata, and visual properties of design elements. Through this interface, it is possible to retrieve component definitions and transform them into structured representations for automated processing. This enables tools to integrate design artifacts directly into development workflows and is the basis for automated analysis of design specifications.

2.1.2 Design-Code Compliance

Design-code compliance refers to the extent to which implemented user interface components align with their corresponding design specifications. In this thesis, compliance is considered at the component level, where properties such as color, spacing, typography, layout, and border radius can be compared between design artifacts and frontend implementations.

The problem is related to architectural or design drift, where the implemented system gradually deviates from its intended specification over time [3]. In the context of design systems, that drift can occur when design artifacts and code implementations are developed independently. Unlike general architectural drift, design-code drift is often small visual and structural properties of reusable interface components.

Some approaches have been created to detect inconsistencies between design specifications and implemented interfaces. Visual regression testing compares rendered screenshots across different versions of an application to detect unintended visual changes. While this is effective for identifying visual differences, these techniques operate on rendered output rather than on structured design specifications and provide limited information into the underlying cause of discrepancies.

Another class of approaches focuses on rule-based design linting or style validation. These tools check code implementations against predefined design tokens or style guidelines, for example by verifying that specific color values or spacing scales are used consistently. Although these methods can enforce certain constraints, they typically rely on manually defined rules and may struggle to capture higher-level relationships between design components and their corresponding implementations.

As a result, ensuring compliance between design artifacts and frontend implementations is still mainly manual. Designers and developers must inspect components across design tools and codebases to identify mismatches, which becomes increasingly difficult as the number of components and variants grows. This creates the need for more automated approaches that can compare structured design data with frontend implementation details.

2.2 Large Language Models

LLMs are neural network models trained on large text corpora to predict and generate natural language. Most modern LLMs are based on the transformer architecture, which uses self-attention mechanisms to model relationships between tokens in a sequence [9]. This architecture helps models capture long-range dependencies and perform a wide range of language-related tasks such as text generation, question answering, summarization, and reasoning.
Recent generations of LLMs have also shown strong capabilities in understanding and reasoning about source code and structured data. Because programming languages share many similarities with natural language, transformer-based models trained on mixed corpora of text and code can learn representations that capture program structure, naming patterns, and relationships between components. Research has shown that LLMs can effectively analyze code, identify inconsistencies, and produce structured explanations of program behavior [5].
Modern LLM systems also support structured output mechanisms that constrain model responses to predefined schemas. Instead of generating free-form text, the model can be required to return outputs that conform to formats such as JSON or other typed structures. This capability is particularly useful when LLMs are integrated into software systems, because it allows generated outputs to be reliably parsed and consumed by downstream components.
Several commercial LLM families are currently used in software engineering workflows, including models in the GPT and Claude families. These models are designed to perform reasoning tasks over complex inputs, including code snippets and structured metadata.They also support mechanisms such as tool calling and structured responses, which allow them to interact with external systems and participate in automated pipelines.
These capabilities make LLMs suitable for tasks that require semantic interpretation of artifacts, such as comparing design specifications with frontend implementations. However, their use in production systems typically requires careful constraints, including schema validation, deterministic preprocessing, and clear task definitions, in order to ensure reliable and reproducible results.

2.2.1 LLMs for Code Understanding

Because modern LLMs are trained on large corpora that include both natural language and programming languages, they can learn representations that capture syntactic structure, naming conventions, and common programming patterns. This enables the models to perform tasks such as code summarization, bug detection, documentation generation, and semantic comparison between code fragments [5].
Unlike traditional static analysis tools that rely on explicit rules or parsers, LLMs can reason about code at a higher semantic level. They are able to interpret the intent of a component, understand relationships between variables and functions, and identify inconsistencies between different representations of the same functionality. This capability is particularly useful when comparing artifacts, such as frontend source code and design specifications, where the information is expressed in differ-

ent formats and abstraction levels.

2.2.2 Prompt Engineering

Prompt engineering refers to the design of instructions and contextual information that guide the behavior of large language models toward desired outputs. Since LLMs are highly sensitive to input phrasing, task framing, and output constraints, prompt design has become an important practical technique for improving reliability, controllability, and consistency in applied systems. In systems that combine conversational interaction with structured downstream processing, prompt engineering is often used to improve answer quality and to reduce hallucinations and create machine-readable outputs.

One common technique is the use of system prompts and role assignment. A system prompt provides high-level behavioral instructions before the user input is processed, for example by defining the model as an assistant specialized in a certain domain or task. Assigning these roles can keep the model focused on relevant behavior. In practical systems, system prompts are often also used to define tool-usage rules, scope boundaries, style constraints, and safety restrictions. This is very useful in agentic systems, where the model must decide when to answer directly and when to call external tools.

Another important technique is the use of few-shot examples. Few-shot prompting supplies the model with one or several input-output demonstrations inside the prompt, showing the desired reasoning pattern or response format. Rather than learning new parameters, the model conditions on these examples at inference time. Few-shot examples are especially useful when tasks require subtle distinctions, domain-specific judgment, or consistent formatting. Prior work has shown that examples can significantly improve performance on structured reasoning and extraction tasks by making the intended behavior more concrete [10]. In practice, few-shot prompting is often combined with carefully phrased instructions to steer the model toward the correct handling of the task.

A third technique is the use of structured output schemas. When LLM outputs must be used by software components, free-form natural language is often not good enough because it can be inconsistent or difficult to parse reliably. To address this, prompts can explicitly request output in JSON or another predefined format. Modern frameworks allow developers to define schemas in code, for example using Pydantic models, and bind these schemas to the model through function calling or equivalent structured output mechanisms. In this way, the schema specifies the expected fields, types, and constraints, while the API-level function definition defines valid responses. This reduces formatting errors and makes the outputs easier to validate automatically. If structured decoding fails, systems often fall back to prompt-based JSON instructions together with schema validation after generation.

Prompt engineering also frequently includes constraint injection, where the prompt explicitly states rules the model must follow during analysis. These constraints may define acceptable tolerances, normalization procedures, grouping rules, or instructions about what should not be reported. These constraints are important in comparison and compliance tasks, where the model must work deterministically over

provided data instead of generating unsupported statements. Anti-hallucination instructions, such as requiring the model to only use explicitly available information and to explain its reasoning with reference to the input data, are a common example of this design strategy.

2.2.3 LLM-Based Agents

An LLM agent is a software system where a pre-trained language model functions as the central decision-making component and external tools that enable the LLM to interact with other systems [11]. In comparison to a regular one-shot LLM application, an agent typically use an iterative loop. It receives an input or observation, reasons about the current state of the task, decides whether an external action is needed, calls a suitable tool, uses the tool output as a new observation, and repeats this process until it can return a final response. This makes the agent suitable for tasks that require multiple dependent steps rather than a single completion.
An LLM agent usually consists of three main elements: the LLM, a set of tools, and an orchestration mechanism. The LLM interprets the user request, selects appropriate actions, and synthesizes intermediate results into a output. The tools provide access to external capabilities such as API calls, database queries, file retrieval, or code execution. The orchestration layer handles the control flow by maintaining state, routing tool calls, feeding results back to the model, and determining when the interaction should terminate.
A key characteristic of LLM agents is their ability to ground model outputs in external data and system functionality rather than relying only on the model's internal knowledge. In practice, tools are exposed to the model through different interfaces, often with structured schemas and descriptive metadata. The model can then generate a structured tool call specifying which function to call and with what arguments. After execution, the result is returned to the agent as a new observation that informs subsequent reasoning. This mechanism enables agents to perform goal-directed, multi-step interaction with software systems in a controlled and auditable way.

2.2.4 The ReAct Pattern

Reasoning and Acting (ReAct), is an agent design pattern in which reasoning steps and external actions exist within a single control loop [11]. Instead of separating planning and execution into distinct phases, the model reasons about the current task, chooses an action, observes the result, and continues from the updated context. This structure makes the agent able to adapt its behavior dynamically as new information becomes available.
The main advantage of ReAct is that it combines thinking and execution in a combined loop. This makes it good at open-ended tasks where the number of required actions may vary from one request to another. Some user requests may require only a single tool call, while others may require several dependent actions in sequence. Because the agent decides step by step whether further action is necessary, it can handle both simple and compound requests without requiring a fixed workflow.

ReAct is also good in settings where tool outputs may influence subsequent decisions. Because the model does not commit to a complete plan in advance, it can adapt if a tool returns unexpected data, partial results, or an error. In practice, this makes ReAct a flexible and robust pattern for applications where requests are interactive, state-dependent, and not fully predictable at the outset.
For the system developed in this thesis, the ReAct pattern is appropriate because user requests may involve a variable number of operations, such as retrieving data, listing components, and running comparison or reporting steps in sequence. A reactive loop therefore offers a simpler and more natural control structure than architectures that require an explicit plan to be generated before execution begins.

2.2.5 LangChain and LangGraph

LangChain is an open-source framework for building LLM-powered applications[12]. It has reusable abstractions for core components such as model interfaces, prompt construction, structured outputs, and tool definitions. One of its main advantages is that it has the same programming interface across different model providers, which makes the integration of external LLM services into an application much easier.
In LangChain, tools are typically defined as typed Python functions with descriptive metadata. These definitions make it possible for the model to call external functionality through structured arguments instead of through unstructured text. LangChain also supports prompt templating and output parsing which provide a good foundation for building agent-like systems.
LangGraph provides a graph-based runtime for handling stateful agent workflows [13]. LangChain primarily provides abstractions for the individual building blocks of an LLM application but LangGraph focuses on execution flow, state management, and iterative control. In LangGraph, an application is represented as a graph of computational nodes connected by transitions over shared state. This representation is very useful for agent systems in which the workflow may involve repeated cycles between model reasoning and tool execution.
This graph-based system is perfect for ReAct-style agents. Reasoning and tool execution can be represented as separate nodes, and the system transitions between them until a termination condition is met. In addition to supporting iterative loops, LangGraph is designed for stateful and long-running workflows. This makes it useful for applications that require persistence, streaming, or controlled execution across multiple steps.
Another useful thing in the LangChain/LangGraph ecosystem is runtime context propagation. Contextual information can be passed at invocation time and be available in the execution graph without the need for global variables. This makes the separation cleaner between core agent logic and integration-specific concerns like external callbacks, user metadata, or session-dependent configuration. Such mechanisms improve modularity and make the system easier to test, maintain, and extend.
In this thesis project, LangChain and LangGraph provide the implementation framework for exposing tools to the model and handling the iterative agent loop. LangChain supports tool binding and model interaction, while LangGraph provides the control structure needed to execute repeated reasoning and action steps in a transparent

and manageable way.

2.3 Component Matching Techniques

Component matching refers to the process of identifying which frontend implementation corresponds to a given design component. In design-code compliance work, this step is necessary before the actual comparison can be performed. If a Figma component is matched to the wrong frontend component, later compliance results will be misleading even if the comparison method itself works correctly. This makes matching an important part of any automated workflow that compares design artifacts with code implementations.

A simple approach is exact name matching, where two components are considered to match only if their names are identical. However, this is often not sufficient in real design systems. Design components and frontend components often follow different naming conventions, contain prefixes or suffixes, use different casing, or describe the same concept with slightly different terms. For example, a component name in Figma may describe a visual or design-system concept, while the corresponding frontend component name may reflect its implementation structure. Therefore, more flexible matching techniques are needed.

One common approach is string similarity matching. String similarity methods compare two names at the character level and produce a score that represents how similar they are. For example, Python's `SequenceMatcher` can be used to compare two sequences and estimate their similarity based on matching subsequences [14]. This type of method can identify names that are almost equal or only differ by small changes, such as spelling, casing, or word order. However, character-level similarity can be limited when two components have different names but still represent the same user interface element.

Another approach is token-based matching. Instead of comparing the complete name as one string, the name is first split into meaningful tokens. This can include splitting on spaces, hyphens, underscores, or camelCase boundaries. Noise words or less informative terms can also be removed. The similarity between two token sets can then be measured using a metric such as Jaccard similarity, which compares the size of the intersection between two sets with the size of their union. Token-based matching is useful when two component names share important words but differ in ordering or formatting. For example, PrimaryButton and ButtonPrimary may have low exact string similarity but high token overlap.

Fuzzy matching combines these ideas by accepting approximate matches instead of requiring exact equality. A fuzzy matching score can be based on character-level similarity, token-level similarity, or a weighted combination of both. This makes fuzzy matching efficient and reproducible, since the same inputs will always produce the same scores. It is also useful as a first filtering step because it can quickly identify likely matches across a large number of components. However, fuzzy matching is still mainly based on surface-level name similarity. It can fail when two components are semantically related but have different names, or when two names look similar but refer to different types of components.

Semantic matching address this limitation by considering the meaning or function

of the components rather than only their names. In the context of this thesis, large language models can be used for semantic matching because they can reason about component names, descriptions, and surrounding context. Instead of only asking whether two strings are similar, the model can be asked whether two components appear to represent the same user interface concept. This can help in cases where naming conventions differ between Figma and the frontend codebase. However, semantic matching also introduces uncertainty because LLM outputs may vary and can be affected by prompt design, available context, and model behavior.
Hybrid matching approaches combine deterministic and semantic techniques. A deterministic fuzzy matching step can first be used to identify clear matches based on name similarity. Components that remain unmatched can then be passed to a semantic matching step, where an LLM can reason about less obvious correspondences. This type of hybrid approach balances efficiency, reproducibility, and flexibility. The deterministic stage handles simple cases in a controlled way, while the semantic stage is reserved for cases where name similarity alone is not enough.

2.4 Related Work

Several existing tools and research areas address parts of the design-code consistency problem. These include visual regression testing, design-system documentation tools, style validation tools, design handoff tools, and recent work on LLM-based software engineering. However, these approaches usually focus on either rendered visual output, code-level rules, or design-to-development communication. Fewer approaches combine design data, frontend code, component matching, LLM-based comparison, and workflow integration in one automated pipeline.
Visual regression testing is one common approach for detecting unintended user interface changes. Tools such as Percy and Chromatic compare screenshots of rendered interfaces across versions and highlight visual differences [15, 16]. This is useful for identifying visual regressions before code changes are merged. However, visual regression testing usually operates on rendered screenshots rather than structured design specifications. As a result, it can show that two rendered states differ, but it does not directly explain whether the implementation is compliant with the original Figma component or which design-system property caused the mismatch.
Another related category is component documentation and isolated UI testing. Storybook is commonly used to develop and document frontend components in isolation, and it can be combined with visual testing services such as Chromatic [17]. This supports frontend quality assurance and makes it easier to inspect different component states. However, Storybook-based workflows mainly focus on the implemented component library. They do not automatically retrieve the corresponding design components from Figma or compare implementation details against design specifications.
Rule-based linting and style validation tools also address consistency in frontend development. For example, Stylelint can enforce CSS rules and conventions, and it can be extended with custom rules for project-specific style requirements [18]. Similar approaches can be used to enforce design tokens, approved color values, or spacing scales. These tools are deterministic and useful for enforcing known

constraints, but they depend on predefined rules. They are less suitable when the relationship between a design component and its implementation requires semantic interpretation, such as understanding conditional styling, component variants, or differences in naming conventions.

Design handoff tools and Figma integrations provide another related area. Figma supports developer workflows by exposing design properties, component metadata, and code-related information through its interface and APIs. Figma Code Connect also allows teams to connect components in a codebase with components in Figma's Dev Mode [19]. These tools improve communication between designers and developers and can reduce friction during implementation. However, they mainly support handoff and navigation between design and code. They do not by themselves provide an automated compliance-checking pipeline that retrieves both sides, matches components, detects inconsistencies, and reports the results through a collaboration interface.

There is also related work on LLM-based software engineering. Large language models have been applied to source code understanding, code generation, bug detection, program repair, and explanation of software behavior [5]. These capabilities are relevant because design-code compliance requires reasoning over frontend implementation details, design properties, and structured data. However, general LLM-based code analysis does not directly solve the design-code compliance problem. The model must be constrained to compare specific properties, avoid unsupported claims, return structured outputs, and operate on data retrieved from both design and code sources.

LLM agents are also relevant to this thesis because they allow language models to interact with external tools and perform multi-step workflows. Agent patterns such as ReAct combine reasoning and action, allowing a model to decide when to call tools, inspect results, and continue the task based on new observations [11]. This is useful for design-code compliance because a user request may require several dependent operations, such as listing components, retrieving data, matching components, running a comparison, and generating a report. However, agent frameworks alone do not define how design-code compliance should be performed. They provide the orchestration mechanism, while the domain-specific pipeline still needs to define extraction, matching, comparison rules, output schemas, and reporting.

3
Methodology

This chapter describes the research methodology used to design, implement, and evaluate the system. The work used a design-oriented research process in which a software system was developed iteratively and evaluated through both functional testing and user feedback.

3.1 Research Approach

The thesis used a Design Science Research (DSR) approach. DSR is good for research where the goal is to design, build, and evaluate a system that addresses a practical problem [8].
The research problem was an existing industrial need. At Ericsson, UX designers and frontend developers work with representations of the same design system components, but these representations exist in different systems and formats. Because these components are worked on separately, inconsistencies can appear over time. The purpose of the research was to study this problem theoretically and to construct and evaluate a working prototype that could support design-code compliance checking in practice.
The research process had four main phases. First, a literature review was conducted to understand the existing work related to design systems, design-code consistency, LLM-based code analysis, prompt engineering, and agent-based LLM systems. The literature review used Google Scholar and arXiv as the main search sources. Search terms included combinations of *design-code consistency*, *design systems*, *LLM code analysis*, *LLM agents*, and *ReAct agents*. The review gave the theoretical foundation for the system design and helped identify relevant technical concepts, such as structured LLM outputs, tool-based agents, and hybrid deterministic and semantic matching strategies.
Second, an internal study of the Ericsson development context was made. The initial problem was provided through the thesis context, but additional discussions with frontend developers and UX designers was used to better understand the workflow, the structure of the component library, and which types of inconsistencies were most relevant to detect. These discussions helped guide the scope of the system and influenced what functionality should be prioritized in the prototype.
Third, the system was developed iteratively through three minimum viable product (MVP) versions. The first MVP focused on establishing a basic end-to-end workflow, including a Slack-based interaction model and early tool creation. The second MVP improved the system with data extraction, component matching, and first compari-

son functionality. The third MVP received the most development effort and focused on improving robustness, polishing features, reducing hallucinations, improving the quality of generated reports, and implementing feedback from user testing.

The system was evaluated with a combination of functional evaluation and user evaluation. The functional evaluation examined if the system could retrieve relevant component data, match design and code components, detect known inconsistencies, and generate structured outputs. The user evaluation focused on if the Slack interface, reports, and explanations were understandable, useful, and appropriate for the workflows of frontend developers and UX designers.

3.2 Development Process

The development process was an iterative and prototype-driven approach. Because this thesis investigated the creation and use of an AI-supported design-code compliance workflow, it was important to continuously test the system against realistic examples and change the implementation based on these tests.

The work began by exploring the technical environment, including Figma, GitLab, Slack, LangChain/LangGraph, and the available internally approved LLM infrastructure. This phase focused on understanding how design components and frontend components could be accessed, represented, and compared. After this the system was developed incrementally through three MVP stages.

The first MVP established the basic system structure. The goal was to verify that a Slack-based interface could connect to backend functionality through an LLM-powered agent and receive responses in the same communication environment used by developers. This version focused on the interaction flow and basic tool execution.

The second MVP had more complete data handling and comparison functionality. Like retrieving component data from Figma and GitLab, normalizing the information into structured JSON representations, and developing initial matching and comparison logic. At this stage the system was tested on selected components to identify whether the retrieved data contained enough information for meaningful compliance analysis.

The third MVP focused on polishing and evaluation. This version included improvements based on repeated testing and user feedback, including clearer report formatting and summaries, more optimized prompts, structured output validation, improved handling of wrong matches, and more reliable tool behavior. Most of the development effort was spent in this stage because many of the challenges appeared after the system was tested on real and controlled component examples.

Testing was performed continuously during development. Each new feature was manually tested after implementation and larger system tests were performed by running the pipeline on selected components with known differences. These tests were used to examine whether the LLM detected actual inconsistencies, missed important differences, or hallucinated issues that were not supported by the input data. When failures happens, prompts, input representations, tolerances, context size or tool logic were adjusted and tested again.

3.2.1 Requirements Gathering

The requirements were gathered from three main sources: the initial thesis problem formulation, discussions with Ericsson developers and UX designers, and feedback from user testing.

The initial problem was provided as part of the thesis description at Ericsson. The need was to investigate whether AI could help detect and explain inconsistencies between UX design components and frontend implementations. This established the high-level requirement that the system should compare design artifacts from Figma with frontend code components and communicate the result in a way that supports collaboration between designers and developers.

Discussions with frontend developers were used to understand the practical aspects of the problem. These discussions helped clarify what types of mismatches were relevant, how frontend components were structured, and what kind of output would be useful in practice. For example, the system needed to support comparison of properties such as colors, spacing, typography, layout, and component variants. It also needed to handle cases where naming conventions differed between Figma and the frontend codebase.

User feedback was gathered throughout the development process. There were six users, mainly frontend developers but also UX designers used and reviewed the system during development. Some users participated in several rounds of testing. Feedback was collected both informally and through a more structured feedback format using Google Forms. This feedback influenced the design of the Slack interface, the structure of generated reports, and the level of detail included in explanations.

Based on this process, the main requirements were:

- The system should get and structure component data from both Figma and GitLab.
- The system should match Figma components to corresponding frontend components, even when naming conventions differ.
- The system should detect and explain differences in relevant visual and structural properties.
- The system should reduce unsupported LLM behavior by constraining prompts and enforcing structured outputs.
- The system should present results through Slack in a format that is understandable for both frontend developers and UX designers.
- The system should support iterative improvement through user feedback and manual correction of incorrect matches.
- The system should meet Ericsson security requirements by using approved internal infrastructure and LLM services.

3.2.2 Technology Selection

The technology selection was chosen based on the use case, the research goals, and the need to integrate with existing development workflows. Since the thesis focused on comparing design specifications with frontend implementations, the system had to interact with the tools already used by the organization.

Figma was selected as the source of design data because it is used to define UX components, variants, and visual properties. The Figma REST API was used to retrieve structured design data programmatically. GitLab was selected as the source of frontend implementation data because the relevant component code is stored in a GitLab-hosted repository. The GitLab integration made it possible for the system to retrieve and update local copies of frontend components for analysis.
Slack was selected as the user interface because frontend developers and designers already use it as a communication channel. A Slack chatbot makes it possible for users to request comparisons, receive reports, and interact with the system without opening a separate tool. This also supports the research goal of investigating how compliance checking can be integrated into existing collaboration workflows.
LangChain and LangGraph was selected for LLM orchestration and agent control. These frameworks gave abstractions for tool definitions, structured model interaction, and ReAct-style agent workflows. This made them suitable for building a system where an LLM can interpret user requests, decide which tools to call, and combine tool outputs into a final response.
An internally approved LLM hosted through Ericsson's Azure-based infrastructure was used for semantic matching, compliance checking, and explanation generation. This choice was needed because the system uses proprietary design and code data. The use of an approved internal LLM service ensured that the thesis complied with Ericsson's data governance and security requirements. Since the model itself could not be fine-tuned, the system relied on prompt design, deterministic preprocessing, structured input representations, schema-constrained outputs to control model behavior.
Python was used for backend development because of its strong ecosystem for API integration, LLM orchestration, data processing, and rapid prototyping. JSON was used as the central representation for design and code components.

3.3 Data Collection

The design data was stored in Figma, while the corresponding implementation data was stored in a GitLab-hosted frontend codebase.
The Figma data consisted of design components defined as component sets with multiple variants. These variants represent different configurations or states of the same component, as visual themes, interaction states, or size variations. The frontend data consisted of the corresponding component implementations in the codebase.
Some of the component data also contained known inconsistencies or issues that had either been identified by developers or UX designer, or observed during development of the system. These cases were useful because they represented realistic examples of the type of design-code drift that the system was intended to detect.

3.4 Evaluation Method

The evaluation was designed to examine both the technical behavior of the implemented system and its usefulness for intended users. Since the system consists of

several connected parts, including data extraction, component matching, LLM-based compliance checking, report generation, and Slack-based interaction, the evaluation was divided into functional evaluation, experimental configuration testing, system performance testing, and user evaluation.
The functional evaluation focused on if the system performed the intended technical tasks correctly. This included testing if component data could be extracted from Figma and GitLab, if Figma components could be matched to frontend implementations, if known design-code inconsistencies could be detected, and if the generated outputs followed the required structured schema. The user evaluation focused on if the Slack interface, reports, and explanations were understandable, useful, and relevant for frontend developers and UX designers.
The evaluation used both real industrial component data and controlled test cases. The real data was used to evaluate the system in the context it was designed for, while the controlled examples made it possible to test the system against known differences. This combination was used because the system needed to work on realistic component structures while also being tested against cases where the expected result was known.

3.4.1 Functional Evaluation

The functional evaluation was performed through repeated test runs on selected components. Most components were chosen because their differences was already known. By using examples with known ground truth, the system could be evaluated based on if it detected actual mismatches, missed relevant differences, hallucinated differences, or reported issues that were not present.
The evaluation examined many aspects. First, the extraction process was evaluated by checking if relevant Figma and frontend component data was retrieved and represented in a usable format. Second, the matching process was evaluated by inspecting if Figma components were correctly linked to their corresponding frontend implementations. Incorrect matches were especially important because they could lead to misleading compliance results.
Third, the compliance checking process was evaluated by comparing the system output against known differences. The analysis focused on categories such as colors, spacing, typography, layout, dimensions, and missing or inconsistent properties. Both small and large differences were included in the test cases to evaluate if the system handled different levels of differences.
Fourth, LLM behavior was evaluated by observing hallucinations, unsupported claims, wrong classifications, and failure cases. A hallucination was when the system reported a mismatch or explanation that was not supported by the provided design or code data. Wrong classifications included cases where the model marked a component as compliant despite known differences, or where it reported a wrong match or mismatch incorrectly. These cases were analyzed and tested to understand if failures were caused by insufficient input data, ambiguous component structure, prompt limitations, model reasoning errors, or incorrect matching.
Latency was also measured as part of the functional evaluation. Since the system is intended to be used interactively through Slack, response time affects usability. The

evaluation therefore considered how long different operations took, including synchronization, matching, compliance analysis, and report generation. Longer-running operations were analyzed to identify trade-offs between output quality, reasoning depth, prompt size, and user experience.

3.4.2 Experimental Setup and Metrics

The functional evaluation was based on a manually verified ground truth set of design-code inconsistencies. The ground truth contained known mismatches between Figma components and frontend implementations. These mismatches were either identified from existing component data or introduced in controlled component variants created for evaluation. The ground truth was used as the reference when evaluating whether the system correctly detected inconsistencies.

The compliance-checking results were evaluated at the inconsistency level. Each reported issue was compared against the ground truth and classified as a true positive, false positive, or false negative. A true positive was a reported inconsistency that corresponded to a known inconsistency in the ground truth. A false positive was a reported inconsistency that was not supported by the ground truth or by the provided design and code data. A false negative was a known inconsistency that the system failed to report.

Precision, recall, and F1-score were used to measure inconsistency detection performance. Precision measured how many of the reported inconsistencies were correct, while recall measured how many of the known inconsistencies were detected. F1-score was used as a combined measure of precision and recall. These metrics were used because the system should both detect real inconsistencies and avoid reporting false issues.

The component matching stage was evaluated by examining whether Figma components were linked to the frontend implementations. A match was successful when the system made a correct match between a Figma component and a frontend component with sufficient confidence. Components that did not reach the required confidence threshold were left unmatched. Incorrect matches were also tracked, since they could lead to misleading compliance results. When the compliance-checking stage classified a pair as a wrong match, the match was removed from the stored matching state and recorded as rejected. This made it possible to evaluate the initial matching output and the effect of the self-correcting wrong-match mechanism.

Several experimental configurations were tested to evaluate how different design choices affected the compliance checking results. The tested configurations had different input representations, different LLM models, and different prompting strategies. The input representation comparison examined the difference between using raw Figma JSON and filtered JSON. The model comparison examined the performance of the available model variants. The prompting comparison examined the difference between a single-pass comparison prompt and a multi-pass prompting strategy where different visual and structural attributes were checked separately. The same ground truth was used when comparing these configurations so that the results could be compared consistently.

The evaluation also considered practical system behavior. Data reduction was mea-

sured by comparing the size of the raw Figma API response with the filtered representation used for compliance checking. Latency was measured by tracking the elapsed time for operations such as synchronization, matching, compliance analysis, and report generation. Token usage was recorded from LLM calls where available, since prompt size and completion size affected both latency and practical usability. The end-to-end Slack workflow was evaluated by testing whether a user could request an analysis, receive progress updates, and obtain a generated report through the Slack interface.

3.4.3 User Evaluation

The user evaluation was conducted with six participants, mainly frontend developers, but also including UX-related participants. Some participants interacted with the system or reviewed its outputs multiple times during the development process. The purpose was to evaluate If the system was useful and understandable from the perspective of its intended users.
Feedback was collected in two ways. First, informal feedback was gathered during iterative testing and demonstrations. This allowed users to comment freely on the system behavior, report format, Slack interaction, and usefulness of the detected issues. Second, a more structured feedback process was conducted using Google Forms. This provided a consistent way to collect user impressions across participants.
The user evaluation focused on the following aspects:

- If the Slack interface was easy to understand and use.
- If the generated reports were clear and well structured.
- If mismatch explanations were understandable.
- If users trusted the system output.
- If the level of detail was appropriate for developers and UX designers.
- If the tool could fit into existing design and development workflows.
- Which parts of the system users found most useful or confusing.

User feedback was used as final evaluation data and also as input to the iterative development process. Feedback from earlier MVP versions influenced later changes to the interface, report structure, explanations, and feature prioritization. This made the evaluation formative as well as summative: it helped improve the artifact during development and also provided evidence about its usefulness at the end of the project.

3.5 Ethical Considerations

The thesis was conducted in an industrial setting and involved proprietary Ericsson design and code artifacts. Because of this, security and data handling were very important ethical and practical considerations during the project.
Only internally approved LLM services were used for processing proprietary data. The selected LLM was hosted through Ericsson-approved Azure infrastructure, which ensured that design and code data were handled according to internal data gover-

nance requirements. No external, unapproved LLM services were used for analysis of proprietary components.
The system stored data only within Ericsson-approved infrastructure. Component data, extracted representations, intermediate files, reports, and related artifacts were handled within Ericsson's internal cloud environment. Access to the system was restricted to authorized users, and the prototype was designed for internal use rather than public deployment.
The system was also designed to support human oversight. Since LLMs can produce incorrect outputs, hallucinations, or misleading explanations, the tool was not treated as an automatic source of truth. Instead, it was designed as a support tool for developers and UX designers. The generated reports are intended to help users identify possible inconsistencies, but final judgment remains with humans. This is especially important because design-code compliance can involve contextual decisions where small differences may be intentional or acceptable.
Another ethical consideration is transparency. The system reports detected mismatches and explanations in a structured way so that users can inspect why an issue was flagged. The use of structured outputs, deterministic preprocessing, and explicit mismatch categories helps make the system behavior more traceable than a purely free-form LLM response.
The project did not involve personal or sensitive user data beyond feedback from participating users. User feedback was used to improve and evaluate the prototype, but the focus of the thesis was on system functionality and workflow usefulness rather than individual user behavior. Since the system is limited to internal software development support, it does not directly affect end users of Ericsson products.
Finally, the environmental impact of the project was limited to prototype-scale development and testing. The system used cloud-based LLM inference and infrastructure, but the scale of experimentation was small compared to production AI systems. The main sustainability consideration was therefore to avoid unnecessary repeated LLM calls where possible, for example by using deterministic preprocessing, batching comparisons, and limiting LLM reasoning to tasks where semantic interpretation was needed.

4

System Design and Implementation

4.1 System Overview

Figure 4.1 shows a simplified overview of the implemented system architecture. The system is accessed through Slack, where a user submits a query to the chatbot. The query is forwarded to the agent, which acts as the central orchestration component. Based on the user request, the agent decides which tools need to be invoked and combines the results into a final Slack response or report.

The figure simplifies the implementation into three main tool groups. The Figma/GitLab tool group retrieves, cleans and structures design data from Figma and frontend implementation data from GitLab. The matching tool group matches corresponding design and frontend components and stores accepted matches in the database. The compliance-checking tool compares matched components and detects differences in visual and structural properties. The database is used as shared persistent storage between these tool groups, allowing extracted components, matches, and generated analysis results to be reused across later requests.

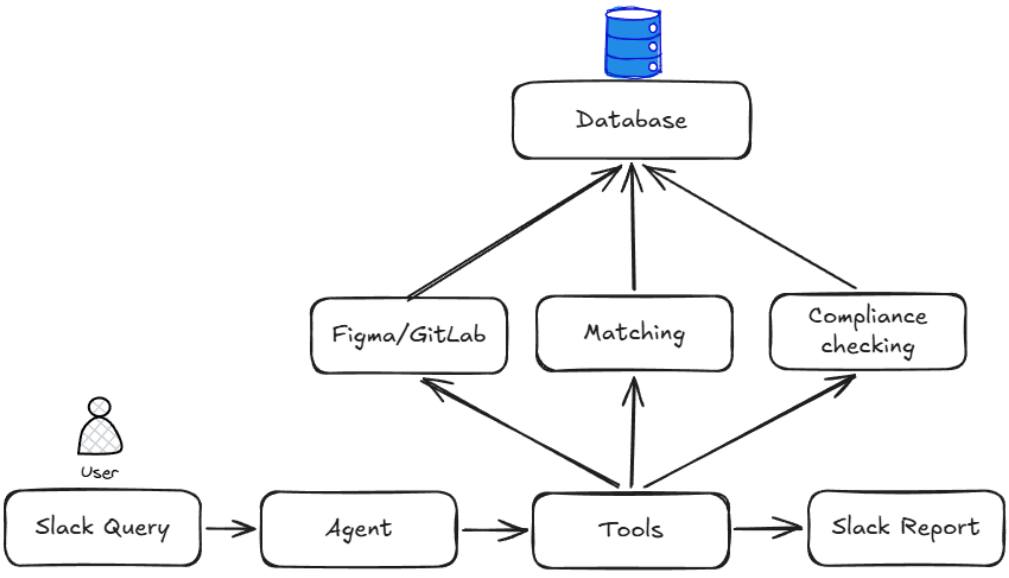

Figure 4.1: Simplified overview of the system architecture and main tool interactions.

The high-level flow is:

`User Query (Slack)` $\rightarrow$ `Slack Interface` $\rightarrow$ `Agent` $\leftrightarrow$ `Tool call(s)` $\rightarrow$ `Response (Slack)`

The system is built around a single LLM-powered agent that interprets natural language requests and orchestrates a set of tools to carry out the design-code compliance workflow. These tools handle data synchronization from Figma and GitLab, component matching between design artifacts and frontend implementations, multi-stage compliance analysis, and support functions such as listing components, retrieving component details, and manually correcting matches.

All user interactions happens through a Slack chatbot. Users can issue simple requests, such as asking for available components, or combine multiple operations in a single message. The agent determines the required sequence of actions and calls the appropriate tools. To keep the data up to date, a background scheduler periodically synchronizes both the Figma and GitLab sources.

4.2 Controlled LLM Orchestration

The core architectural principle of the system is controlled integration of LLMs within a deterministic pipeline. Rather than relying on a single LLM prompt, the system separates deterministic processing from semantic reasoning and limits LLM usage to pre-defined subtasks.

The architecture follows these design principles:

1. Deterministic preprocessing is performed whenever possible.
2. LLM reasoning is restricted to clearly scoped semantic decisions.
3. All LLM outputs are constrained using structured schemas.

This design mitigates hallucination, improves robustness, and increases reproducibility in an enterprise environment where compliance decisions must be traceable and consistent.

4.3 Agent Design

The system uses a single ReAct-style agent that alternates between LLM reasoning steps and tool calls until a final response is made. The agent instance and LLM client are instantiated once and reused across calls. This reduces repeated authentication and gives a consistent configuration of model parameters for all tool calls.

4.3.1 Streaming and User Feedback

Agent execution is processed incrementally instead of a single blocking call. Before each tool call, the system shows a human-readable progress message to Slack. This improves transparency during long operations such as repository synchronization or compliance analysis which helps reduce perceived latency and as a status update for the user.

4.3.2 Slack Tool Integration

Tools are designed to be independent of the Slack SDK. The Slack interface injects callback functions into the agent execution context which allows tools to upload reports or send progress updates without directly depending on Slack-specific libraries. For local testing without slack, these callbacks are disabled and tools fall back to local file storage. This design improves modularity and makes testing easier.

4.4 Tool Design

The agent has seven tools available. Each tool is a standalone function with a single responsibility and operates on structured JSON representations of design and frontend components.

The tools can be grouped into four categories:

1. **Data Synchronization:** Tools responsible for extracting and normalizing component data from GitLab and Figma. Raw components are converted into structured JSON representations before any semantic analysis is performed.
2. **Component Matching:** A two-phase hybrid algorithm is used. Phase 1 applies deterministic fuzzy matching using a composite similarity score combining normalized string similarity and token overlap. Phase 2 applies LLM-based semantic matching only to unresolved components. The mapping of matches supports one-to-many relationships and respects previously rejected or user-corrected pairs.
3. **Compliance Checking:** For each matched pair, structured JSON representations are compared using LLM-based analysis prompts. When a frontend component corresponds to multiple Figma variants, batch comparison is performed in a single LLM call to reduce API round-trips. The LLM returns structured results that classifies each comparison as compliant, non-compliant, wrong match, or error. Detected wrong matches are automatically removed from the mapping state to prevent repeated misclassifications.
4. **Inspection and Manual Correction:** These tools provide component listings, lookup functionality, and manual reassignment of incorrect matches. User corrections remain in future matching runs, ensuring that human feedback improves future automation.

This layered tool design makes sure that LLM reasoning is applied only after deterministic preprocessing and that decisions are always grounded in structured data.

4.5 Prompt Design

All prompts are centrally defined and version-controlled to ensure consistency and traceability. The system uses three primary prompt categories: agent routing prompts, component matching prompts, and compliance analysis prompts.

4.5.1 Agent Routing Prompt

The agent system prompt defines its role as a compliance assistant and specifies explicit tool-routing rules. It restricts the agent from making unnecessary recommendations and ensures the use of exact component names from the stored dataset. This reduces hallucinated tool arguments and enables more predictable tool calls.

4.5.2 Component Matching Prompt

The matching prompt works only on unmatched components after deterministic fuzzy matching. It instructs the LLM to match based on functional similarity instead of name similarity and to return structured output containing explicit confidence scores. Only matches that the LLM is confident are correct are accepted and added to the component mapping.

4.5.3 Compliance Analysis Prompt

The compliance prompt first verifies that a matched pair represents the same type of UI element based on content. If not, the model must return a `WRONG_MATCH` classification. For valid matches, the analysis is structured into predefined categories (colors, spacing, typography, layout), with explicit tolerance rules.
All compliance outputs are constrained using Pydantic-based structured schemas via function-calling mechanisms. This eliminates free-text parsing and makes sure that every LLM response uses a predefined JSON format.

4.6 Component Data Pipeline

Before matching or compliance analysis can happen, the system must get and normalize data from two sources: the Figma design file and the frontend codebase. Each source has their own APIs, formats, and abstractions. The data pipeline transforms both into a JSON representation stored in the database, which allows tools to operate on a consistent, source-independent interface.
Extraction is separated from analysis. By treating synchronization as an own phase triggered by the user, the system makes sure that the input to the matching and compliance checking work deterministically with respect to the extracted data.

4.6.1 Figma Data Extraction

The Figma extraction stage retrieves design component specifications via the Figma REST API and converts them into structured JSON artifacts for automated comparison.
The system goes through the document tree of the configured Figma file and collects all nodes of type `COMPONENT_SET`, which represent components with multiple variants like theme variations or interaction states. Pages that names contain the substring "icon" are excluded, as icons are images with no comparable metadata and outside the scope of this thesis.

The extraction pipeline extracts only the visual properties needed for comparison. For each component set, the individual variants are retrieved with their rendered properties such as colors, spacing, typography, and layout. The extraction applies less detail at deeper nesting levels, capturing full style information for the top-level variant nodes while limiting deeper children to only color and typography data. Non-visual elements like vector shapes and drawing primitives are skipped entirely, and null values are stripped from the output. This filtering reduces the data size by 40-60% while preserving everything needed for the compliance check.
For each variant, the extractor gets a subset of CSS-comparable properties, including:

- Background and border colors (normalized and converted to hexadecimal form)
- Corner radius
- Padding values (top, right, bottom, left)
- Layout properties (layout mode, spacing, alignment)
- Dimensions (width and height).

Raw Figma responses contain a lot of metadata (effects, constraints, plugin data) which are not relevant to compliance analysis. By keeping only properties with CSS counterparts, the system reduces token consumption and limits the LLM reasoning to visually important discrepancies.

4.6.2 Frontend Code Extraction

The frontend extraction stage retrieves the component library from GitLab and converts each component into a structured JSON representation.
The repository is cloned or updated locally to get the latest updates. The extractor over the local repository while ignoring internal or deprecated components.
For each component, relevant source files are collected, including:

- The component template defining the markup.
- Base style definitions containing structural properties.
- Theme-specific style overrides with design token definitions.
- Shared design tokens for spacing, typography, and colors.

License headers are stripped during preprocessing to avoid unnecessary token consumption during later LLM analysis.

4.7 Component Matching

Compliance analysis requires associating each Figma component with its corresponding frontend implementation. This mapping is difficult due to different naming conventions and many-to-one relationships.
The system uses a hybrid two-phase strategy that combines deterministic similarity scores with LLM-based semantic reasoning.

4.7.1 Phase 1: Deterministic Fuzzy Matching

The first phase computes similarity scores between all Figma and frontend component names using a composite metric:

1. Character-level similarity based on normalized subsequence matching.
2. Token-level Jaccard similarity over meaningful name tokens.

The final similarity score is a weighted combination (40% character similarity, 60% token overlap). Greater weight is given to token overlap, because meaningful design-system tokens has more semantic information than raw character alignment. Pairs go over a threshold are accepted as matches.
This deterministic stage resolves straightforward associations without the use of the LLM.

4.7.2 Phase 2: Constrained LLM Semantic Matching

Components still not matched after the fuzzy phase are passed to the LLM for semantic matching. The model receives the full set of frontend component names, existing matches for context, and the list of unmatched Figma components. It is instructed to match based on functional similarity instead of name similarity.
Matches are returned in structured form using a Pydantic schema, including a confidence score and reasoning summary. Only matches with confidence ≥ 0.7 are accepted. This threshold balances recall and precision while preventing low-confidence semantic guesses.

4.8 Compliance Checking

After component data has been extracted and corresponding Figma and frontend components have been matched, the system performs compliance checking. The purpose of this stage is to determine whether a frontend implementation follows the visual and structural specification defined in Figma. The comparison is performed at the component level and uses the structured JSON representations created by the earlier data pipeline.
The compliance checker receives a matched pair consisting of one or more Figma component and one frontend component. The Figma side contains extracted design properties such as colors, spacing, typography, layout information, border radius, and dimensions. The frontend side contains the component markup, styling definitions, theme-specific overrides, and shared design token references. These inputs are passed to an LLM-based comparison prompt that is constrained to analyze only the provided data.
The compliance checker is implemented as a separate tool in the agent system. This keeps the comparison logic separated from the Slack interface and from the agent routing logic. The agent decides when the compliance tool should be called, while the tool itself handles the comparison, structured output validation, wrong-match handling, and report generation.

4.8.1 Comparison Strategy

The comparison strategy is based on controlled LLM reasoning over structured input data. Instead of asking the model to freely inspect arbitrary design and code

artifacts, the system provides filtered JSON representations that contain only information relevant to design-code comparison. This reduces unnecessary context and limits the model's attention to properties that can be compared between Figma and the frontend implementation.
For each matched pair, the compliance prompt instructs the model to compare the Figma specification against the frontend implementation. The model is not asked to judge the overall quality of the component or suggest unrelated improvements. Instead, it must determine whether the implementation is compliant with the design specification based on predefined comparison categories.
When a frontend component corresponds to several Figma variants, the variants can be compared in batch within one LLM call. This reduces the number of API calls and avoids repeating the same frontend context for every variant. The output still contains separate comparison results for the relevant variants, which makes it possible to inspect which specific state or configuration contains an inconsistency.
The compliance checker classifies each comparison into one of four main outcomes: compliant, non-compliant, wrong match, or error. A compliant result means that no relevant difference was found within the checked properties. A non-compliant result means that one or more design-code inconsistencies were detected. A wrong match means that the Figma and frontend components do not appear to represent the same type of UI element. An error means that the comparison could not be completed because of missing data, invalid input, or another failure in the comparison process.

4.8.2 Match Validation

Before property-level analysis is performed, the system validates whether the matched pair appears to represent the same UI component. This step is important because component matching is not guaranteed to be correct.
The LLM is instructed to first inspect the component names, available structure, and provided content to determine whether the pair is functionally comparable. If the pair appears to represent completely different UI elements, the model returns a `WRONG_MATCH` classification instead of continuing with property-level comparison.
When a `WRONG_MATCH` result is returned, the system automatically removes the pair from the stored matching state and records it as rejected. This creates a feedback loop between compliance checking and component matching. Incorrect matches discovered during compliance analysis are not reused in later runs, which reduces repeated false comparisons and improves the matching state over time.

4.8.3 Property Analysis and Tolerance

If the pair passes the match validation step, the model continues with property-level analysis. The comparison focuses on visual and structural properties that are relevant for design-system compliance. The main categories are colors, spacing, typography, and layout. Border radius and dimensions are also considered when they are explicitly available in the extracted data.
Color comparison includes properties such as background color, text color, and border color. Colors are normalized during preprocessing, and the compliance prompt

instructs the model to treat colors as exact matches only after normalization. This is necessary because small differences in color values can represent real design-system inconsistencies.
Spacing comparison includes padding, margin, and related layout spacing values when available. Typography comparison includes properties such as font size, font weight, line height, and related text styling. Layout comparison includes structural properties such as layout direction, alignment, spacing between elements, dimensions, and border radius. For spacing, typography sizes, dimensions, and border radius, the comparison allows a tolerance of (± 1) pixel. This tolerance was chosen as a strict but practical threshold. Because pixels are the smallest visual unit represented in the extracted design and implementation data, a difference of one pixel can often be caused by rounding, rendering, or conversion differences between Figma values and frontend CSS values. Reporting every one-pixel difference would produce findings that are technically different but visually insignificant. The tolerance is intentionally kept small because larger differences in spacing, dimensions, typography, or border radius may affect visual consistency in a design system.
Detected issues are categorized based on the type of discrepancy. A `MISMATCH` is used when both the design and frontend implementation define a comparable property, but the values differ beyond the accepted tolerance. A `MISSING` issue is used when a property is present in the Figma design but cannot be found or inferred from the frontend implementation. This distinction makes the report more useful because it separates incorrect values from absent implementation details.

4.8.4 Structured Output Enforcement

All compliance outputs are constrained using Pydantic-based schemas and function-calling mechanisms. The schema defines the expected fields, classifications, mismatch categories, explanations, and supporting values that must be returned by the model. This prevents the compliance checker from relying on free-form text parsing.
Structured output enforcement is important because the comparison results are used by later parts of the system. The report generator, Slack upload logic, wrong-match correction mechanism, and evaluation scripts all depend on machine-readable results. By forcing the LLM to return a predefined structure, the system can validate the output before it is stored or presented to the user.
If the model output does not satisfy the expected schema, the comparison is treated as an error instead of being silently accepted. This makes failures explicit and prevents bad responses from being interpreted as valid compliance results. The structured schema also improves traceability because each detected inconsistency is represented with a category, description, and the design and implementation values when available.

4.8.5 Report Generation

After the comparisons have been completed, the system combines the structured results into a report. The report includes a summary of the analyzed component or component set, the overall compliance status, detected inconsistencies, wrong

matches, and errors. For each detected issue, the report describes the affected property category and explains the difference between the Figma specification and the frontend implementation.
Reports are serialized to a database using unique filenames. This avoids collisions when multiple users or comparison requests are handled close to each other. The use of files also prevents large JSON results from being passed back through the agent context window, which would increase token usage and make the agent response harder to manage.
When the compliance checker is called through Slack, the generated report is uploaded directly to the Slack conversation. After upload, the local report file can be removed. When the system is executed outside Slack, for example during local testing, the report is kept locally instead. This separation makes the compliance tool usable both in the deployed Slack workflow and during development or debugging.

5
Results

This chapter presents the results of the implemented system. The results show that the system can retrieve design and code artifacts, represent them in a structured format, match corresponding components, and generate structured reports of design-code inconsistencies through a Slack-based interface.
To make the results interpretable, this chapter distinguishes between two types of evaluation data. The first dataset was used for compliance checking and had manually established ground truth. In this context, ground truth refers to a manually verified list of expected inconsistencies between selected Figma components and their corresponding frontend implementations. The ground-truth set consisted of 10 selected components with 139 variants in total. Within this set, 33 manually verified inconsistencies were used as the ground truth for evaluating inconsistency detection. For these components and variants, the expected inconsistencies were known in advance through manual inspection of the design and code representations.
An inconsistency refers to a case where a design property in Figma did not match the corresponding implementation in the frontend code. The evaluated inconsistency types included visual and structural properties such as colors, spacing, padding, gaps, typography, border radius, layout properties, alignment, dimensions, and component variant differences. A detected inconsistency was counted as correct if it corresponded to one of the manually verified ground-truth inconsistencies. A missed inconsistency was counted as a false negative, while an unsupported reported issue was counted as a false positive.
The second dataset was used for component matching and did not have the same type of manually verified inconsistency ground truth. Instead, it consisted of the broader extracted component library, containing 189 components that required matching between Figma and the frontend codebase. This dataset was used to evaluate how many likely component correspondences the matching pipeline could identify across the larger component set.

5.1 Component Data Pipeline Results

The implemented data pipeline successfully retrieved component data from both Figma and the frontend codebase. The Figma extraction pipeline identified component sets and variants from the configured design file and transformed them into structured JSON representations. The extracted design data included comparison relevant properties such as colors, border radius, padding, layout mode, spacing, alignment, and dimensions. Non-relevant metadata, null values, and system-

generated fields were all successfully removed during preprocessing.

The frontend extraction pipeline retrieved component implementations from the GitLab-hosted repository and converted the relevant source files into structured component representations. For each frontend component, the system collected the component markup, styling definitions, theme-specific style overrides, and shared design token references. This made it possible to compare Figma design properties with the frontend implementation even when the final styling was distributed across several files.

A main result of this stage is that both design artifacts and frontend code could be represented in a common JSON-based format suitable for LLM-based comparison. The filtered representation also reduced unnecessary input size while preserving the information needed for design-code compliance checking.

5.2 Component Matching Results

The component matching stage was evaluated on the full set of extracted design and frontend components. At the time of evaluation, the dataset contained 189 components that required matching between Figma and the frontend codebase. Since the underlying component set was actively maintained during the evaluation period, the matching pipeline was executed three times to account for minor changes in the extracted data. Across these runs, the combined fuzzy matching and LLM-based semantic matching pipeline produced 109 confirmed matches in the representative final run. The remaining components could not be matched with sufficient confidence and were left unmatched.

The matched components represented the cases where the system found a likely correspondence between a Figma component and a frontend implementation. The unmatched components mainly consisted of cases where no sufficiently confident match was produced by either the deterministic fuzzy matching stage or the LLM-based semantic matching stage. These components were not passed forward as confirmed matches in the matching state.

When the full pipeline was executed with the self-correcting wrong-match mechanism enabled, the system also identified incorrect matches during the compliance checking stage. In the representative final run, 10 of the initially accepted matches were classified as WRONG_MATCH. These cases were automatically removed from the stored matching state and recorded as rejected matches, preventing the same incorrect component pairs from being reused in later runs.

The final matching result therefore consisted of three groups: successfully matched components, unmatched components, and matches that were initially accepted but later rejected through the WRONG_MATCH correction mechanism. The results should therefore be interpreted as the observed output of the matching pipeline for the component set available at the time of evaluation, rather than as fixed values for a static dataset.

5.3 Compliance Checking Results

The compliance-checking performance was evaluated using precision, recall, and F1-score. Precision measures how many of the reported inconsistencies were correct, while recall measures how many of the known ground-truth inconsistencies were detected by the system. The F1-score combines precision and recall into a single metric and is defined as: $F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$. In this thesis, the F1-score is interpreted as an overall measure of inconsistency detection quality. A high F1-score means that the system both detects most known inconsistencies and avoids reporting many unsupported issues. For the intended workflow, a low recall would be problematic because important design-code inconsistencies would be missed, while low precision would reduce trust in the tool by producing too many false alarms. These metrics were calculated on the 33 manually verified inconsistencies in the compliance-checking ground-truth set, not on the full set of 189 components used for matching.
An F1-score above 0.90 is considered strong for this prototype because it indicates that the tool could support designers and developers by highlighting likely inconsistencies with few missed issues or false positives. Scores around 0.75–0.85 may still be useful for exploratory analysis, but would require more manual review. Scores close to or below 0.60 are considered weak for this workflow, since the tool would either miss too many inconsistencies or produce too many unreliable findings.
The strongest evaluated configuration used filtered JSON, GPT-5.1, and multi-pass prompting. In this configuration, the system achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510. These results indicate that the system was able to detect most of the manually verified inconsistencies while keeping the number of false positives low.

Table 5.1: Best observed inconsistency detection performance.

Configuration	Precision	Recall	F1-score
Filtered JSON + GPT-5.1 + Multi-pass prompting	0.9577	0.9444	0.9510

The results also showed that the design representation affected detection performance. Filtered JSON outperformed raw JSON across all evaluation metrics. Raw JSON achieved a precision of 0.9254, recall of 0.8611, and F1-score of 0.8920, while filtered JSON achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510.

Table 5.2: Impact of design representation on inconsistency detection.

Representation	Precision	Recall	F1-score
Raw JSON	0.9254	0.8611	0.8920
Filtered JSON	0.9577	0.9444	0.9510

Model choice also had a clear effect on the results. GPT-5.1 achieved the best overall performance, while GPT-mini produced weaker but still usable results. GPT-nano achieved high precision but very low recall, meaning that it was often correct when it reported an issue, but missed many true inconsistencies.

Table 5.3: Impact of model selection on inconsistency detection.

Model	Precision	Recall	F1-score
GPT-nano	0.9524	0.2778	0.4301
GPT-mini	0.8209	0.7639	0.7914
GPT-5.1	0.9577	0.9444	0.9510

Prompting strategy also had a major impact. The multi-pass strategy, where the comparison was split into separate checks for different attributes, clearly outperformed the single-pass strategy. Multi-pass prompting achieved an F1-score of 0.9510 compared with 0.6066 for single-pass prompting.

Table 5.4: Impact of prompting strategy on inconsistency detection.

Strategy	Precision	Recall	F1-score
Single-pass prompting	0.7400	0.5139	0.6066
Multi-pass prompting	0.9577	0.9444	0.9510

5.4 System Performance and Practical Pipeline Results

This section presents system-level results from the implemented prototype. The results cover data reduction, handling of large component inputs, theme-token extraction, token usage, and the end-to-end Slack workflow. These results are based on prototype testing using selected Figma component data and corresponding frontend implementation data.

5.4.1 Data Reduction Results

The Figma extraction pipeline reduced the size of the design representation by approximately 40–60%. This was achieved by filtering the raw Figma REST API response to include only compliance-relevant component data. The extracted data included component sets, colors, typography, layout information, spacing, dimensions, and selected child-node properties. Null values, empty objects, non-relevant metadata, and non-comparable design information were removed.

The frontend representation was also reduced before comparison. The original frontend data contained template files, component-specific styling, theme styling, shared variables, and global palette definitions. After introducing the slim frontend representation, bulky shared and global SCSS files were replaced with extracted token values while preserving the component template, base styles, component-specific theme file, and resolved tokens. This reduced the frontend data size by approximately 50-85%.

5.4.2 Large Component Handling

Prototype testing showed that large components produced less stable LLM outputs when the comparison task was too broad. For larger components, such as button components with many variants and states, the combined Figma and frontend input could reach approximately 40,000–70,000 characters. In these cases, observed output issues included hallucinated values, missed properties, inconsistent findings across repeated runs, and incorrect grouping of findings under variants.

Two splitting strategies were tested: variant splitting and property-scoped splitting. Variant splitting divided Figma variants into batches and compared each batch separately. This produced duplicate findings and made it difficult to group results across all variants. It also caused misleading use of broad labels such as `allVariants` when only a subset of variants had been checked.

Property-scoped splitting was used in the final pipeline. This approach kept all variants in the input but narrowed each LLM call to one property category. For large components, the comparison was split into focused checks such as colors, padding, gap, typography, and layout. This reduced duplicate findings and allowed the model to group findings across all variants within each property category.

5.4.3 Prompt Structure Results

The prompt structure was adjusted during prototype development. Earlier prompts with many rules, edge cases, and formatting requirements produced less predictable outputs. In later versions, the prompts were made shorter and more focused, with each prompt targeting a specific property category.

The comparison prompts were structured with static component data first and task-specific instructions last. This structure was used for the property-scoped comparison calls. It also allowed repeated calls for the same component to share the same prompt prefix, which enabled prompt caching for the static data portion.

5.4.4 Theme Token Resolution Results

A recurring issue in the frontend data was that many visual properties were defined through CSS variables, SCSS variables, mixins, and theme-specific overrides rather than direct values. In initial comparisons, the LLM sometimes failed to resolve these chains correctly. Failure cases included reporting values as missing, stopping at an intermediate variable name, or resolving to an incorrect value.

The final pipeline used a two-pass approach. In the first pass, a dedicated theme-token extraction call resolved variable chains into final values and stored the results. In the second pass, the compliance checker received the resolved token values together with the component-specific styles and theme overrides.

This approach reduced the number of unresolved variable cases in the comparison input. The comparison data also became more explicit, since final values for colors, spacing, typography, and borders were available directly in the input.

5.4.5 Token Usage and Practical Performance

Token usage varied depending on component size and the number of comparison calls. For a small component, such as a tooltip, one comparison used approximately 25,000 input tokens across three LLM calls and around 500 output tokens. For a larger component, such as a button with many variants, one comparison used approximately 85,000 input tokens across five focused calls and around 3,000 output tokens. A larger suite of 33 components required approximately 500,000–800,000 input tokens in total.

Table 5.5: Approximate token usage for compliance checking.

Comparison scope	Input tokens	Output tokens	Number of calls
Small component, e.g. tooltip	∼25k	∼500	3
Large component, e.g. button	∼85k	∼3k	5
Full suite, 33 components	∼500k–800k	Depends on findings	Multiple

Using the GPT-5.1 API price at the time of writing, the approximate cost was $1.25 per million input tokens and $10.00 per million output tokens. Converted using an exchange rate of approximately 1 USD = 0.87 EUR, this corresponds to about €1.09 per million input tokens and €8.68 per million output tokens. Under this pricing, a small component comparison using around 25,000 input tokens and 500 output tokens costs approximately €0.03, while a larger component comparison using around 85,000 input tokens and 3,000 output tokens costs approximately €0.12. The full 33-component suite, using approximately 500,000-800,000 input tokens, costs approximately €0.54-€0.87 for input tokens alone, with additional cost depending on the number of generated output tokens.
Several optimizations were used to reduce repeated processing. Theme-token extraction results were cached per component. Prompt caching was enabled by keeping the static data prefix consistent across property-scoped calls. A variant cap was also used to prevent components with many variants from producing uncontrolled input sizes.

5.4.6 End-to-End Workflow Results

The implemented prototype completed the full workflow from Slack request to generated compliance report. Through the Slack interface, a user could trigger synchronization of design and code data, run component matching, perform compliance checking, and receive the generated result as a report.
The system provided progress updates during longer operations such as synchronization and compliance analysis. Detailed comparison results were serialized and returned as report files instead of being sent as large messages directly through the Slack conversation. This allowed the system to return structured compliance results while keeping the Slack interaction readable.
The final prototype integrated Figma data extraction, frontend code extraction, component matching, theme-token extraction, LLM-based compliance checking, report generation, and Slack-based interaction into one workflow.

5.4.7 User Evaluation Results

The user evaluation showed that the overall response to the prototype was positive. Participants found the Slack interface easy to use and understand, and the generated explanations were generally considered clear and useful. The Slack-based workflow was also seen as a practical way to access the system, since it allowed users to request analyses and receive results through an existing communication channel.

Several points of feedback led to changes in the implementation. During testing, users asked for clearer feedback during longer-running operations. For example, when the system started synchronization, matching, or compliance checking, users wanted to know that the request had been received and that the system was still working. Based on this feedback, progress updates and time-estimate-style messages were added to the Slack interaction.

Participants also found the JSON reports useful but too detailed to read directly as the main output. The structured JSON was valuable for detailed inspection, but it was considered overwhelming when users only wanted to understand the main issues quickly. To address this, an AI-generated summary was added alongside the JSON report. The summary gives users a shorter explanation of the detected issues, while the JSON report remains available for deeper inspection.

The evaluation also showed that users wanted clearer guidance about what the bot could do. Although the interface itself was considered easy to use, participants suggested that instructions or a short description of the available functionality would make the system easier to understand for new users. A description was therefore added to the Slack bot to explain its purpose and supported actions.

Trust was identified as an important factor in the usefulness of the system. Participants reacted positively when the reported inconsistencies were correct and well explained. However, trust decreased when the system reported too many false inconsistencies, meaning issues that users did not consider to be actual problems. This shows that precision is especially important for this type of tool, since incorrect findings can reduce user confidence even when the system also detects many real issues.

Participants also suggested that the system could be useful outside of manual Slack-based interaction. In particular, several users mentioned that similar checks could be integrated into a CI/CD pipeline or pull request workflow. This would allow compliance checking to run automatically when frontend components are changed, making the system more proactive and helping teams detect inconsistencies earlier in the development process.

Overall, the user evaluation indicated that the system was understandable, useful, and relevant to the intended workflow. The main concerns were related to progress visibility, report readability, user guidance, and trust in the correctness of detected inconsistencies. These findings support the design of the system as a human-support tool rather than a fully automatic source of truth.

5.5 Summary of Results

The results show that the proposed system can support automated design-code compliance checking in an industrial design-system context. The data pipeline was able to retrieve and structure component data from both Figma and GitLab. The matching pipeline combined deterministic fuzzy matching with constrained LLM-based semantic matching, allowing the system to handle both simple name-based matches and more difficult naming differences. The compliance pipeline generated structured reports of detected inconsistencies across visual and structural properties. The best-performing compliance configuration used filtered JSON, GPT-5.1, and multi-pass prompting, achieving a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510. These results indicate that LLM-based design-code inconsistency detection is most effective when the model is supported by filtered input representations, explicit structured output, and task decomposition.

The user evaluation further showed that the Slack interface and generated explanations were understandable and useful, but that progress visibility, concise report summaries, clear bot instructions, and low false-positive rates were important for user trust and practical adoption.

Overall, the implemented system demonstrates that an AI-powered Slack chatbot can be used as an interface for design-code compliance checking. The prototype does not replace designers or developers, but provides a structured assistant that can highlight potential inconsistencies and support communication between the two roles.

6
Discussion

This chapter discusses the results of the implemented AI-powered design-code compliance system. The discussion focuses on what the results mean in relation to the research questions, why certain design decisions were important, what trade-offs were identified, and which limitations exist.

6.1 Analysis of Results

The results show that the implemented system can support a end-to-end design-code compliance workflow. The system was able to retrieve design data from Figma, retrieve frontend implementation data from GitLab, structure both sources into comparable representations, match corresponding components, perform LLM-based compliance checking, and return reports through Slack.
A general pattern across the results is that the system performed best when the task was constrained and structured. The strongest results came from filtered input representations, property-specific prompting, schema-constrained outputs, and a feedback mechanism for correcting wrong matches. This suggests that LLM-based design-code compliance checking should not be treated as a single open-ended generation task, but as a controlled pipeline where the LLM is used for specific reasoning steps.

6.1.1 Component Representation and Data Extraction

A central result is that the system was able to transform two different types of components into representations that could be compared. Figma components are represented as design metadata, variants, layout properties, and visual attributes, while frontend components are represented through implementation code, styling definitions, theme variables, and component logic. These artifacts do not map directly to each other. The results therefore show that the intermediate representation was an important part of the system.
The comparison between raw JSON and filtered JSON shows that the structure of the input had a clear effect on detection performance. Raw JSON achieved a precision of 0.9254, recall of 0.8611, and F1-score of 0.8920, while filtered JSON achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510. The largest improvement was in recall, which means that the filtered representation helped the system detect more of the known inconsistencies.
This suggests that the raw Figma representation contained information that was

unnecessary and also distracting for the LLM. In this task, more input data did not automatically lead to better results. The model performed better when the input was reduced to comparison-relevant properties such as colors, spacing, typography, layout, dimensions, and border-related information.

6.1.2 Component Matching and Wrong-Match Correction

The component matching results show that automatic matching was useful, but also that it remained one of the more uncertain parts of the pipeline. Out of 189 components that required matching, the combined fuzzy and LLM-based matching pipeline produced matches for 109 components. This means that the system could identify a large part of the relevant design-code relationships, but also that a substantial number of components remained unmatched.
This result is important because it shows that matching is a bottleneck for full design-system coverage. However, leaving uncertain components unmatched is preferable to accepting weak matches, since an incorrect match can produce misleading compliance reports.
The wrong-match correction results also show why matching should not be treated as a fixed preprocessing step. When the full pipeline was executed, 10 initially accepted matches were later classified as `WRONG_MATCH` during compliance checking. These matches were removed from the stored matching state and recorded as rejected matches.
This result shows that the initial matching stage was not fully reliable on its own, but also that the pipeline could recover from some incorrect matches. The self-correcting mechanism therefore improved the robustness of the system by preventing the same incorrect mappings from being reused in later runs.

6.1.3 Compliance-Checking Performance

The compliance-checking results show that the best-performing configuration was able to detect most of the manually verified inconsistencies while keeping false positives low. The strongest configuration, using filtered JSON, GPT-5.1, and multi-pass prompting, achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510.
This indicates that the system reached a useful balance between two important risks. Low precision would make the system less trustworthy because users would receive too many unsupported findings. Low recall would make the system less useful because real inconsistencies would remain hidden. The high precision and recall therefore suggest that LLM-based compliance checking can work well when the task is constrained by structured inputs, clear mismatch categories, and schema-based outputs. This also aligns with the user evaluation, where participants indicated that too many false positives would reduce trust in the system. Precision is therefore not only a technical metric in this thesis, but also a practical requirement for adoption in a real workflow.
However, these results should be interpreted together with the matching results. The compliance-checking metrics describe performance on evaluated component pairs with known inconsistencies. They do not mean that the system fully checked

the entire component library automatically. Since many components remained unmatched, full-library compliance checking would still require improved matching, manual mapping, or additional component coverage.

6.1.4 Model Choice and Prompting Strategy

The model comparison shows that model capability had a major influence on the final result. GPT-5.1 achieved the best overall performance, while GPT-mini produced weaker but still usable results. GPT-nano achieved high precision but very low recall, resulting in a much lower F1-score. This means that GPT-nano was often correct when it reported an issue, but missed many true inconsistencies.
For a design-code compliance assistant, this is a serious limitation. A model that only reports issues when it is very certain may appear reliable, but if it misses many actual differences, the component library can still drift over time. The results suggest that smaller models may be useful for narrow or low-risk checks, but that more capable models are needed when the system is expected to detect a broader range of visual and structural inconsistencies.
The prompting strategy had an even clearer effect on the results. Single-pass prompting achieved a precision of 0.7400, recall of 0.5139, and F1-score of 0.6066, while multi-pass prompting achieved a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510.
This difference shows that design-code comparison is too broad to be handled reliably as one general prompt. A component can contain differences in colors, spacing, typography, layout, dimensions, border properties, and variant-specific behavior. When all of these categories are checked at once, the model must reason over many different aspects of the input simultaneously. By splitting the task into separate property-focused checks, the multi-pass strategy reduced the complexity of each comparison and improved both precision and recall.

6.1.5 Large Components and Theme Tokens

The results from large component handling support the same conclusion. Large components were difficult because they produced longer prompts and because they contained many variants, nested structures, and property categories. Splitting by variant reduced input size, but it also caused problems because the model lost the full variant context and could produce duplicated or fragmented findings.
Property-scoped splitting worked better because it preserved the component context while narrowing the comparison task. This shows that task decomposition is more important than simple token reduction. For LLM-based compliance checking, the input should be reduced in a way that preserves the information needed for the specific reasoning task.
The theme-token extraction results also show that the LLM should not be expected to solve all subtasks at once. Frontend styling often depends on chains of design tokens, CSS variables, theme-specific overrides, and implementation-specific style logic. If the model must both resolve these values and compare them against Figma in the same step, the task becomes less reliable.

The two-pass approach, where token values were first extracted or resolved and then used during comparison, reduced this burden. This supports the broader design principle used in the system: deterministic or specialized preprocessing should be used where possible, while the LLM should be used for tasks that require semantic interpretation.

6.1.6 Token Usage and Practical Feasibility

The token usage affects how the system should be used in practice. While the cost increases when the pipeline is run frequently or across a larger component library, the observed per-component cost is still low compared to the time cost of manual inspection by a designer or developer. For important or recently changed components, an automated check may therefore be worthwhile even if it uses a relatively large number of tokens.
This also means that the system is better suited for targeted checks, release preparation, or scheduled design-system audits, rather than continuous execution after every minor change. Input reduction and caching remain important, since filtering component data, resolving theme tokens in advance, and reusing static prompt prefixes help reduce repeated token usage. Overall, token usage does not make the approach impractical, but it should be considered when deciding how often and at what scale the system is run.

6.1.7 System Performance and Slack Workflow

The system-level results show that the prototype was technically feasible, but that scalability remains an important concern. Data reduction reduced the size of the design representation, and the slim frontend representation reduced unnecessary frontend input. These reductions made the LLM calls more practical, but the token usage and latency results still show that full-library analysis can become expensive and slow.
This is especially relevant when using multi-pass prompting, since higher accuracy comes at the cost of more LLM calls. Therefore, a production version would need careful control of synchronization frequency, caching, batching, and possibly incremental comparison of only changed components.
The end-to-end Slack workflow shows that the system can be integrated into an existing collaboration environment. This is important because design-code compliance is partly a technical problem and partly a communication problem. A report that is generated outside the normal workflow may be ignored, while a Slack-based assistant can make the results available where developers and designers already communicate.

6.2 Answering the Research Questions

The overall research question was:

> **RQ1: How can an AI-based system be created and used to detect and explain inconsistencies between UX design components and their corresponding frontend implementations?**

The results show that this system can be created by combining deterministic data processing with constrained LLM-based reasoning. The implemented system retrieves data from Figma and GitLab, converts both sources into structured JSON representations, matches corresponding components, performs compliance checking, and returns structured reports through Slack. The system is not fully automatic in the sense that all results can be trusted without review, but it demonstrates that an AI-based assistant can identify and explain many design-code inconsistencies in a useful and structured way.

6.2.1 RQ1.1: Retrieving and Representing Design and Frontend Components

The first sub-question asked how design components from Figma and frontend components from a GitLab-hosted codebase can be reliably retrieved and represented in a form suitable for AI-based comparison.
The results show that reliable retrieval requires both source-specific extraction and task-specific representation. For Figma, the system extracted component sets, variants, and visual properties such as colors, layout information, spacing, typography, dimensions, and border-related properties. For the frontend codebase, the system extracted component markup, base styles, theme-specific overrides, and shared design-token references. These artifacts were then represented in JSON so that later stages of the pipeline could operate on a consistent data format.
The results also show that retrieval alone was not sufficient. The extracted data had to be filtered and normalized before it became suitable for LLM-based comparison. Filtering the Figma data reduced the design representation by approximately 40–60%, while the slim frontend representation reduced frontend input size by approximately 50–85%. This reduction improved practical system behavior by removing irrelevant metadata and focusing the input on properties that were relevant for compliance checking. The answer to RQ1.1 is therefore that design and frontend components can be represented for AI-based comparison by combining API-based extraction with structured filtering, normalization, and JSON-based representation.

6.2.2 RQ1.2: Matching Figma Components to Frontend Implementations

The second sub-question asked how Figma components can be automatically matched to corresponding frontend implementations when naming conventions differ.
The results show that a hybrid matching strategy is suitable for this problem. Deterministic fuzzy matching handled straightforward cases where names were similar enough to match automatically. LLM-based semantic matching then handled cases where naming differences required interpretation of functional similarity. This combination reduced unnecessary LLM use while still supporting more complex matching cases.
However, the matching results also show that automatic matching remained a bottleneck. Out of 189 components that required matching, the system produced matches for approximately 109 components. This means that the system was able to iden-

tify many relevant design-code relationships, but also that a substantial number of components remained unmatched. This indicates that automatic matching can support the workflow, but that full design-system coverage would still require improved matching logic, manual mapping, or additional component metadata.
The results also show that matching should not be treated as a one-time preprocessing step separate from compliance checking. Approximately 10 initially accepted matches were later classified as `WRONG_MATCH` during the compliance-checking stage. These matches were removed from the stored matching state and recorded as rejected matches. This feedback loop made the matching process more robust by allowing the system to correct some incorrect mappings over time. The answer to RQ1.2 is therefore that Figma and frontend components can be matched using a hybrid fuzzy and semantic approach, but that wrong-match detection and human oversight remain important for reliability.

6.2.3 RQ1.3: Designing a Comparison Pipeline for LLM-Based Detection

The third sub-question asked how a comparison pipeline can be designed to combine structured design data and frontend code representations for LLM-based inconsistency detection.
The results suggest that the comparison pipeline should be decomposed into several controlled stages. First, the input data should be filtered and normalized. Second, frontend token values should be resolved where possible before comparison. Third, the LLM should validate whether the matched pair represents the same component. Fourth, property-level comparison should be split into focused checks, especially for large components.
The best-performing configuration used filtered JSON, GPT-5.1, and multi-pass prompting, achieving a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510. This indicates that LLM-based design-code inconsistency detection can work effectively when the task is structured properly. The weaker results from raw JSON, smaller models, and single-pass prompting show that the performance depends strongly on input representation, model capability, and task formulation.
These results should be interpreted in relation to the matching coverage. The high compliance-checking performance describes the evaluated component pairs and known inconsistencies, not automatic coverage of the entire component library. Since many components remained unmatched, the comparison pipeline depends on the quality of the preceding matching stage. The answer to RQ1.3 is therefore that an effective LLM-based comparison pipeline should not simply send all available data to the model. Instead, it should use a controlled architecture where preprocessing, token resolution, match validation, property-specific prompting, and structured outputs work together.

6.2.4 RQ1.4: Designing a Slack-Based Interface

This sub-question asked how a Slack-based interface can support querying, reporting, and notification of design-code inconsistencies in a clear and helpful way.

The implemented prototype completed the workflow from Slack request to generated compliance report. Users could trigger synchronization, matching, compliance checking, and report generation from the same communication environment. Progress updates during long-running operations made the system more transparent, which is important for an interactive tool where some operations may take time.
The Slack interface helped expose the pipeline in a way that fits existing developer and designer workflows. Instead of requiring users to run scripts or inspect local JSON files, the system returned structured reports through Slack. This supports the idea that compliance checking should be integrated into collaboration workflows and not only into technical build pipelines.
However, the results also show that Slack should mainly be understood as an access layer for the underlying pipeline. Slack is useful for triggering analyses, receiving summaries, and sharing reports, but it is less suitable for inspecting large or complex compliance outputs directly in a chat thread. The answer to RQ1.4 is therefore that a Slack-based interface can support design-code compliance checking by making the workflow accessible in an existing communication tool, but the clarity and usefulness of the interface should be evaluated further through more systematic user feedback.

6.3 Design Decisions and Trade-offs

Several design decisions had a strong influence on the final system behavior. One of the most important was the decision to combine deterministic processing with LLM reasoning. Deterministic processing was used for data retrieval, filtering, normalization, caching, and straightforward matching. LLM reasoning was used for semantic matching, theme-token extraction, and compliance checking. This division was necessary because the task contains both predictable data-processing steps and ambiguous semantic interpretation steps.
A pure rule-based approach would likely be too rigid for this problem because the relationship between Figma components and frontend implementations is not always direct. Naming conventions differ, styling can be distributed across several files, and design intent may be expressed through variants and nested structures. At the same time, a pure LLM-based approach would be too unpredictable and expensive. The hybrid design represents a practical trade-off: deterministic steps provide stability and efficiency, while LLM steps provide flexibility for tasks that require interpretation.
Another important decision was to use structured output schemas. This reduced the risk of unusable free-text responses and made it possible for the system to validate and process LLM outputs automatically. In a prototype or demo, free-text explanations may be acceptable, but in a workflow that generates reports and updates matching state, outputs must be machine-readable. Structured output also makes the system easier to debug because failures can be traced to specific fields and categories.
The use of multi-pass prompting introduced a trade-off between accuracy and cost. Multi-pass prompting improved detection performance massively, but it also required more LLM calls. For small components, this cost may be acceptable. For

large component libraries, repeated calls could become expensive and slow. The token usage results show that this trade-off must be managed through caching, variant caps, selective comparison, and possibly background execution. The system therefore favors reliability over minimal API usage, but this choice may need adjustment depending on deployment constraints.
The property-scoped splitting strategy also involved a trade-off. Keeping all variants in the input allowed the model to group findings across variants, but it meant that each property-specific call still received a large amount of data for complex components. This was chosen because variant splitting produced duplicate and misleading findings. The trade-off is that property-scoped splitting may still be expensive, but the generated results are more coherent.
The Slack interface also involved trade-offs. Slack made the system accessible and easy to integrate into existing communication workflows, but it is not ideal for displaying large or complex reports directly in conversation threads. This was addressed by returning report files instead of long messages as well as a more human readable summary of the report. In future versions, Slack could be combined with a dedicated web dashboard for exploring results in more detail.

6.4 Limitations

The system has several limitations that should be considered when interpreting the results.
First, the evaluation was performed in a specific industrial context with a limited set of components. The system was designed around Ericsson's design system, Figma files, GitLab-hosted frontend codebase, and styling conventions. Although the general architecture could be adapted to other environments, the exact extraction logic, matching assumptions, and comparison prompts may not transfer directly to other design systems or frontend frameworks.
Second, the system compares extracted representations rather than the final rendered user interface. Figma JSON and frontend source files contain useful structural and visual information, but they do not always capture the final runtime appearance. CSS inheritance, browser rendering behavior, dynamic states, responsive behavior, and application logic may affect what the user actually sees. This means that some inconsistencies may be missed, while some reported inconsistencies may not matter in the final rendered interface.
Third, the system depends on the quality and completeness of the extracted input data. If a Figma component does not explicitly define a property, or if a frontend value is hidden behind complex logic or runtime behavior, the LLM may not have enough information to make a reliable judgment. The system can only compare what has been extracted and represented. It does not verify whether the design specification itself is complete, correct, or feasible.
Fourth, the system relies on LLMs, which introduces uncertainty. Even with structured prompts and schemas, LLM outputs may vary, especially for large or ambiguous inputs. The results show that prompt structure, input size, and task complexity all affect reliability. This means that the system should not be treated as an automatic authority. It should be used as a support tool that highlights potential

inconsistencies for human review.
Fifth, the token usage and latency of the system may limit scalability. The prototype can process selected components, but running frequent full-library comparisons could become costly and slow. The current optimizations, such as theme-token caching, prompt caching, and variant caps, reduce this issue but do not remove it completely. A production system would need more advanced scheduling, incremental comparison, and possibly prioritization of changed components.
Sixth, the current system focuses mainly on component-level visual and structural properties. This is appropriate for detecting differences in colors, spacing, typography, layout, and related attributes, but it does not fully address higher-level behavior, accessibility, interaction logic, or complete page-level flows. These aspects are important in real user interfaces but are outside the current scope.
Finally, user evaluation is still limited. The system was designed to support developers and UX designers, but a stronger evaluation would require more systematic feedback from intended users after they have used the tool on real tasks. Without this, the technical results show that the system works as a prototype, but they do not fully prove long-term usefulness in daily development work.

6.5 Future Work

Several future works were identified based on the results and limitations of the prototype. These mainly concern improved component matching, stronger comparison methods, better scalability, deeper workflow integration, and more extensive user evaluation.
One important area for future work is to improve the component matching process. The current system combines deterministic fuzzy matching with LLM-based semantic matching, which made it possible to match many components even when naming conventions differed. However, a large number of components still remained unmatched, and some initially accepted matches were later rejected by the wrong-match correction mechanism. Future versions could improve matching by using additional metadata from Figma and the frontend codebase, import relationships, Storybook metadata, or usage examples.
Another direction is to include rendered UI comparison in addition to extracted design and code representations. The current system compares structured data from Figma and frontend source files, which is useful for identifying differences in properties such as colors, spacing, typography, layout, and border radius. However, the final rendered interface can be affected by CSS inheritance, browser behavior, runtime logic, responsive layout, and application state. A future version could render frontend components in a controlled environment and compare screenshots, computed CSS values against the corresponding Figma specification. This would make it possible to detect inconsistencies that are not visible from static source-code extraction alone.
Scalability is another important area for future work. The prototype showed that LLM-based comparison can produce useful results, but token usage and latency can become high for large components or larger component suites. Future versions could use incremental comparison, where only changed components or changed properties

are analyzed after each synchronization. The system could also use caching more extensively, reuse previous comparison results, and prioritize components based on recent changes, known issues, or user requests. Background jobs and scheduled analysis could reduce the need for users to wait for long-running comparisons inside Slack.

The prompt and model strategy could also be improved. The results showed that filtered JSON and multi-pass prompting improved performance, but these choices also increased complexity and cost. Future work could investigate smaller or specialized models for simpler subtasks, while reserving stronger models for difficult semantic comparisons. Another possible direction is to use a more adaptive comparison strategy, where simple property checks are handled deterministically and the LLM is only called when the system encounters ambiguity. This could reduce latency and cost while keeping the benefits of LLM-based reasoning where it is most useful.

The user interface could also be developed further. Slack was useful because it allowed the system to fit into an existing communication workflow, but Slack is not ideal for exploring large reports or comparing detailed component data. Future versions could combine Slack notifications with a dedicated web dashboard. Slack could be used for triggering analyses and receiving summaries, while the dashboard could provide filtering, search, visual comparison, historical results, manual match correction, and detailed inspection of detected inconsistencies.

Another future direction is deeper integration with development workflows. Instead of only being triggered manually through Slack, the system could be integrated with pull requests, CI/CD pipelines, or scheduled design-system checks. For example, when a frontend component is changed, the system could automatically compare it against the corresponding Figma component and post a report in the relevant pull request or Slack channel. This would make compliance checking more proactive and allow inconsistencies to be detected earlier in the development process.

7
Conclusion

This thesis investigated how an AI-based system can be designed and used to detect and explain inconsistencies between UX design components and their corresponding frontend implementations. The work was motivated by the practical challenge of maintaining design-code consistency in large industrial design systems, where design artifacts and frontend implementations are maintained separately by designers and developers. To address this problem, a proof-of-concept system was developed in an industrial context at Ericsson.

The implemented system combines several parts into one workflow. It retrieves design component data from Figma, retrieves frontend implementation data from a GitLab-hosted codebase, transforms both sources into structured JSON representations, matches corresponding components using a hybrid fuzzy and LLM-based matching approach, performs LLM-based compliance checking, and returns the results through a Slack chatbot interface. The system was designed as a controlled pipeline where deterministic preprocessing, structured representations, schema-constrained outputs, and task-specific prompts are used to make LLM behavior more reliable and traceable.

The results show that this type of system can support automated design-code compliance checking. The data pipeline was able to extract and represent relevant information from both Figma and frontend code. The matching pipeline was able to produce matches for 109 of 189 components, while leaving uncertain components unmatched rather than forcing weak correspondences. When the full pipeline was executed, the wrong-match correction mechanism identified and rejected 10 initially accepted incorrect matches. This shows that automatic matching can support the workflow, but also that matching remains one of the more difficult and uncertain parts of the problem.

The compliance-checking results show that LLM-based comparison can be effective when the task is carefully constrained. The strongest evaluated configuration used filtered JSON, GPT-5.1, and multi-pass prompting, achieving a precision of 0.9577, recall of 0.9444, and F1-score of 0.9510 on the evaluated inconsistency detection task. These results indicate that the system was able to detect most known inconsistencies while keeping unsupported findings low. The comparison between raw and filtered JSON also showed that more input data does not automatically improve LLM performance. Instead, the model performed better when the input was reduced to properties relevant for comparison, such as colors, spacing, typography, layout, dimensions, and border-related information.

The thesis also shows that task decomposition is important for LLM-based compliance checking. Multi-pass prompting, where different visual and structural proper-

ties are checked separately, performed better than a single broad comparison prompt. Similarly, theme-token extraction and property-scoped comparisons reduced the burden on the model by separating value resolution from inconsistency detection. These results support the broader design principle used in the system: deterministic or specialized preprocessing should be used where possible, while the LLM should be used for semantic reasoning tasks that are difficult to solve with simple rules.

The Slack-based interface demonstrated that compliance checking can be integrated into an existing collaboration workflow. Users could trigger synchronization, matching, comparison, and report generation through the same communication platform already used by developers and designers. This is important because design-code compliance is not only a technical problem but also a communication problem. The generated reports and progress updates made the results accessible without requiring users to interact directly with the underlying pipeline.

The overall research question can therefore be answered as follows: an AI-based system for design-code compliance can be created by combining structured data extraction from design and code sources, hybrid component matching, controlled LLM-based comparison, schema-constrained outputs, and workflow integration through a collaboration interface such as Slack. Such a system can detect and explain many inconsistencies between design components and frontend implementations, but it should be treated as a support tool rather than a fully automatic source of truth.

Several limitations remain. The system was evaluated in a specific industrial context and depends on the structure of Ericsson's design system, Figma files, GitLab repository, and frontend styling conventions. The system also compares extracted representations rather than the final rendered user interface, which means that runtime behavior, browser rendering, responsive layout, and application logic are not fully captured. In addition, the system depends on the quality of the extracted data and on LLM behavior, which can still vary for large or ambiguous inputs. Token usage and latency also limit scalability, especially for full-library analysis. Finally, the user evaluation was limited, and more systematic long-term evaluation would be needed to understand how useful the tool is in daily development work.

Future work should focus on improving component matching, increasing scalability, and strengthening evaluation. Matching could be improved through better use of component metadata, manual mapping interfaces, historical corrections, or embeddings. Scalability could be improved through incremental comparison, prioritization of changed components, and tighter integration with development workflows such as pull requests or CI/CD pipelines. The system could also be extended to include rendered UI comparison, accessibility checks, interaction behavior, and page-level flows. A larger user study with developers and UX designers using the tool on real tasks would also be valuable for evaluating trust, usability, and practical usefulness over time.

In conclusion, the thesis demonstrates that LLM-based design-code compliance checking is feasible and useful when implemented as a controlled and structured pipeline. The main contribution is not only the use of an LLM, but the combination of deterministic preprocessing, hybrid matching, structured comparison, self-correction, and Slack-based reporting into an end-to-end workflow. The prototype shows that AI can help developers and designers identify design-code drift

more efficiently, while still requiring human review for the final judgment.

Bibliography

[1] Y. Lamine and J. Cheng, "Understanding and Supporting the Design Systems Practice," *arXiv preprint arXiv:2205.10713*, 2022. Available: https://arxiv.org/abs/2205.10713.

[2] K. J. K. Feng, T. W. Li, and A. X. Zhang, "Understanding Collaborative Practices and Tools of Professional UX Practitioners in Software Organizations," *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*, 2023. doi: 10.1145/3544548.3581273. Available: https://doi.org/10.1145/3544548.3581273.

[3] N. Ali *et al.*, "Architecture Consistency: State of the Practice, Challenges, and Research Directions," *Empirical Software Engineering*, 2018. doi: 10.1007/s10664-017-9515-3.

[4] S. Zhang, T. Zhang, J. Cheng, and S. Zhou, "Who Is to Blame? A Comprehensive Review of Challenges and Opportunities in Designer–Developer Collaboration," *Proceedings of the ACM on Human-Computer Interaction*, 2025. https://dl.acm.org/doi/abs/10.1145/3711105

[5] H. Jelodar, M. Meymani, and R. Razavi-Far, "Large Language Models (LLMs) for Source Code Analysis: Applications, Models, and Datasets," https://arxiv.org/abs/2503.17502

[6] Figma, "Documentation," Figma Developer Docs, 2026. https://developers.figma.com/docs/rest-api/

[7] Figma, "Free Design Handoff Tool for Designers & Developers," Figma, 2026. https://www.figma.com/design-handoff/

[8] A. Dresch, D. P. Lacerda, and J. A. V. Antunes Jr., "Design Science Research," in *Design Science Research: A Method for Science and Technology Advancement*, Springer, Cham, 2014, pp. 67–102. https://doi.org/10.1007/978-3-319-07374-3_4

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017. Available: https://arxiv.org/abs/1706.03762.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020. Available: https://arxiv.org/abs/2005.14165.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," *International Conference on Learning Representations*, 2023. Available: https://arxiv.org/abs/2210.03629.

[12] LangChain, "LangChain Documentation: Overview," LangChain Docs, 2026. Available: https://docs.langchain.com/oss/python/langchain/overview.

[13] LangChain, "LangGraph Documentation: Overview," LangChain Docs, 2026. Available: https://docs.langchain.com/oss/python/langgraph/overview.

[14] Python Software Foundation, "difflib — Helpers for computing deltas," Python Documentation. Available: https://docs.python.org/3/library/difflib.html.

[15] BrowserStack, "Visual Testing with Percy," BrowserStack Percy Documentation, 2026. Available: https://www.browserstack.com/docs/percy/overview/visual-testing-basics.

[16] Chromatic, "Visual Testing for Storybook," Chromatic Documentation, 2026. Available: https://www.chromatic.com/storybook.

[17] Storybook, "Visual Tests," Storybook Documentation, 2026. Available: https://storybook.js.org/docs/writing-tests/visual-testing.

[18] Stylelint, "Stylelint: A Mighty CSS Linter," Stylelint Documentation, 2026. Available: https://stylelint.io/.

[19] Figma, "Code Connect," Figma Developer Documentation, 2026. Available: https://developers.figma.com/docs/code-connect/.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY