Universiteit Leiden

AI-driven Smart User Interface Generation from UMLs for Model-Driven Engineering

Ruike Yuan (s3789659)

Supervisors:
Guus Ramackers & Jan van Rijn

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 24, 2026

Abstract

Model-driven engineering for user interface generation has evolved through three broad phases. Classical approaches (2000–2015) applied model-based transformation rules to convert software models directly into UI artifacts [CCT+03]. Low-code platforms (2015–2022) preserved this model-driven principle within proprietary environments, improving productivity but limiting flexibility [ACM25]. Since 2023, LLM-based UI generation has shifted focus toward Prompt-to-UI pipelines that produce visually flexible interfaces from natural language. However, these approaches rely on prompts rather than software design models, limiting the reuse of formal interaction semantics and reducing traceability between design models and generated interfaces. This thesis proposes that model-driven engineering is most valuable to serve as software semantics and constraint layer. In this approach, software design models (UML) provide the structured interaction semantics from which the web-pages and sections of the UI specification are derived. LLM then serves as a refinement mechanism, enabling users to iteratively revise the generated UI specification (JSON) through natural language instructions. The refined UI specification then serves as the input to Jinja templates for generating a runnable full-stack software prototype, including both backend and frontend according to the UI specification (JSON).

Acknowledgement

I would like to thank Lisa Tweedie, PhD, from Tweedie Consulting UK for her suggestions and comments.

Statement on the Use of Generative AI

During the preparation of this thesis, the author used AI language models to support idea development for layout and content design, and to correct grammar mistakes. The author directed this process, and reviewed all AI-assisted content.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
  - 1.1.1 Research Problem . . . 1
- 1.2 Research Goals and Hypotheses . . . 1
  - 1.2.1 Research Objective . . . 1
  - 1.2.2 Hypotheses . . . 1

2 Background and Related Work **2**
- 2.1 Model-Driven Engineering (MDE) . . . 2
- 2.2 UML Models in Software Development . . . 2
  - 2.2.1 Class Diagrams . . . 2
  - 2.2.2 Use Case Diagrams . . . 2
  - 2.2.3 Activity Diagrams . . . 2
- 2.3 Large Language Models (LLMs) in Software Development . . . 2
- 2.4 Low-Code and No-Code Development . . . 3
- 2.5 Comparison with WordPress . . . 3
- 2.6 Intermediate Representations and DSLs . . . 3
- 2.7 Object-Oriented User Interface (OOUI) Principles . . . 4
- 2.8 Human-in-the-Loop (HITL) Approaches . . . 4
- 2.9 The AI4MDE Framework . . . 4
- 2.10 Summary and Research Gap . . . 4

3 Research Approach and Methods **5**
- 3.1 Methodology . . . 5
- 3.2 Input Data . . . 5
- 3.3 LLM-based UI Generation . . . 6
- 3.4 Human-in-the-Loop Iteration . . . 7
- 3.5 Verification and Evaluation . . . 7

4 System Implementation **7**
- 4.1 LLM Interaction Overview . . . 8
- 4.2 Architecture Overview . . . 9
- 4.3 Metadata Extraction and UML Mapping . . . 11
  - 4.3.1 YAML Transformation Rule Set . . . 12
- 4.4 UI Candidate Generation and Regeneration . . . 15
  - 4.4.1 Prompt Design . . . 16
  - 4.4.2 Human-in-the-Loop for Candidate Regeneration . . . 18
- 4.5 Validation and Guardrails . . . 21
- 4.6 Rendering and Prototype Generation . . . 22
- 4.7 Interface Metadata DSL . . . 22
  - 4.7.1 Page and Section Schema . . . 22
  - 4.7.2 Design System Tokens . . . 23
- 4.8 Summary . . . 25

5 Evaluation **25**
5.1 Evaluation Setup . . . 25
5.2 General Usability . . . 25
5.3 Detailed Evaluation Dimensions . . . 27
5.4 Perceived Utility . . . 28
5.5 Qualitative Feedback . . . 28

6 Conclusion **28**
6.1 Summary of Contributions . . . 28
6.2 Answers to Research Hypotheses . . . 30
6.2.1 H1: OOUI Principle Reflection and Traceability . . . 30
6.2.2 H2: Enterprise Application Suitability . . . 30
6.2.3 H3: Human-in-the-Loop Effectiveness . . . 30
6.3 Limitations . . . 31
6.4 Future Work . . . 31

A LLM Prompt Templates **33**
A.1 Base Styling Context (Regeneration) . . . 33
A.2 Regeneration Diversity Constraints . . . 33

References **34**

1 Introduction

1.1 Problem Statement

1.1.1 Research Problem

Automated interface generation in model-driven engineering aims to reduce development effort by deriving user interface artifacts directly from software design models [Dri20, vA22]. However, existing template-based model-driven approaches primarily rely on predefined transformations from individual design models to interface artifacts [Dri20, vA22]. Although they exploit structural and behavioral information explicitly represented in UML models, they provide limited support for synthesizing interaction semantics across class, use case, and activity diagrams into a coherent, actor-oriented interface specification. For example, the semantics captured in class diagrams are typically processed independently from those in use case and activity diagrams, limiting the ability to derive a coherent interaction model from multiple UML views.
Another increasingly popular approach to generate full stack software with UI is to use LLMs that produce visually flexible interfaces from natural language prompts, but they require formal design knowledge to be translated into prompts rather than directly leveraging existing software models. As a result, the generated user interfaces of software are weakly connected to the underlying system specification, reducing traceability, consistency, and the reuse of modelled design knowledge.

1.2 Research Goals and Hypotheses

1.2.1 Research Objective

The objective of this research is to design, implement, and evaluate a model-driven approach that systematically derives an actor-scoped, object-oriented user Interface Metadata specification from UML class, use case, and activity diagrams through formal inference rules, and to investigate whether this specification can (1) make actor scope, domain-object structure, and workflow logic explicit and traceable to their source UML elements; (2) serve as a structured, constraining input that grounds LLM-based full-stack prototype generation within the existing AI4MDE pipeline [Ver25, vD24]; and (3) support human-in-the-loop refinement through natural-language edits to the Interface Metadata DSL that trigger live prototype regeneration.

1.2.2 Hypotheses

- **H1:** A formal rule set based on object-oriented UI principles can transfer UML class, use case, and activity diagrams into user interface specification, providing a traceable, actor-scoped foundation for LLM-based UI generation.
- **H2:** AI-generated Interface Metadata can provide prototype-level support for multi-user, workflow-driven enterprise applications.
- **H3:** Human-in-the-loop feedback can iteratively refine generated user interfaces and align them more closely with designer intent.

These hypotheses are evaluated through a small-scale user study with nine participants. Participants answer a questionnaire regarding usability, actor support, clarity of the workflow and the diversity of the generated UI candidates, after using the system.

2 Background and Related Work

2.1 Model-Driven Engineering (MDE)

Model-Driven Engineering is a software development methodology that uses high-level models as the primary input to the development process [Dri20, vA22]. Instead of manually writing code, system structure, behavior, and interactions are captured in models designed by developers. Compared with manual coding, MDE can improve productivity and consistency, and enables the automatic generation of software artifacts such as code, documentation, and tests [Dri20, vA22]. In practice, MDE frameworks and tools convert these models into software prototypes or even fully functional applications [vA22].

2.2 UML Models in Software Development

The Unified Modeling Language (UML) is a modeling language widely applied in designing software. It can be applied in MDE to provide structured visualizations of system components (Class Diagram), interactions (Use Case Diagram) [Wil24], and workflows (Activity Diagram) [Ver25].

2.2.1 Class Diagrams

Class diagrams describe the structure of a system by showing the system's classes, their attributes, methods, and the relationships between objects, serving as the foundation for generating data structures and object-oriented components in software [Dri20].

2.2.2 Use Case Diagrams

Use case diagrams capture requirements from the perspective of system actors by describing how users (actors) interact with the system to achieve specific goals (use cases). It is important in software design as it represents user roles, permissions, and system boundaries [Wil24].

2.2.3 Activity Diagrams

Activity diagrams show workflows, processes, and state transitions within a system. The sequence of actions, decision points, parallel executions, and synchronization between activities are all described by activity diagrams, making them useful for specifying dynamic behavior and guiding interface interactions [vA22].

2.3 Large Language Models (LLMs) in Software Development

Large Language Models (LLMs) are increasingly used in software engineering tasks such as code generation, requirements interpretation, debugging, modelling support, and user interface generation [HZL+24, CTBV23, WSL+24]. Their ability to interpret natural language and produce

structured output makes them useful for design tasks where multiple valid solutions may exist [CSC25]. Recent UI generation work also shows growing interest in using LLMs and automated feedback to generate interface code [WSL+24].
However, LLMs can also produce incorrect or unsupported outputs, especially when domain constraints are not explicitly enforced [DMJ+24, CTBV23]. For this reason, recent LLM-based systems often use agentic architectures, guided prompts defined in structured schema, and validation mechanisms instead of generating final artifacts directly [YZY+23, DMJ+24].

2.4 Low-Code and No-Code Development

Low-code and no-code platforms allow users to build software with little or no manual programming. They typically provide reusable components, visual editors, workflow builders, and data connectors, thereby raising the abstraction level of software development and reducing the technical barrier for non-programmers [ACM25]. Recent research also shows increasing interest in combining low-code interaction with LLMs, for example by allowing users to edit and confirm structured generation workflows through graphical interfaces [CMW+24]. However, low-code and no-code platforms still face limitations such as platform lock-in, scalability challenges, governance concerns, and usability issues [ACM25, PAA23]. This thesis differs from general low-code platforms by focusing on UML-derived model context, actor-specific pages, workflow-aware sections, and OOUI-oriented interface generation.

2.5 Comparison with WordPress

WordPress is a complete open source website creation platform, which can quickly create websites through themes, plug-ins and page editors. However, the prototype of WordPress page is set in the management of theme and content, not made from the software design mode such as UML. Therefore, the relationship between software model, user interface, actors, domain objects and workflow cannot be clearly expressed, and it cannot be traced back later. In contrast, in the method proposed in this paper, interface DSL is made directly from UML model. In this way, we can trace the relationship between the software design model and the created user interface, and at the same time, we can make the interface better. In this way, in the initial stage of software development, this method is more in line with the requirements of consciously making models and confirming.

2.6 Intermediate Representations and DSLs

Intermediate representations serve the purpose of bridging the gap between abstract models and concrete implementation artifacts in model-driven engineering [Fow10, Dri20]. For example, when used for the purpose of UI generation, a DSL can provide an abstract description of a page, components, data binding, navigation and styling before it is turned into code ready to be rendered into a functional interface. Generated interfaces can more readily be tested, manipulated, translated, or otherwise altered [Fow10].
LLM-based UI generation research in recent years has re-affirmed the advantage of employing structured intermediate representations.
For example, the SpecUI approach of [CSC25] makes use of SPEC (a hierarchical and parametrized representation) derived from a reference screenshot to specify a UI and reports significant improve-

ment over prompt-based baselines with regard to generation fidelity and controllability. This thesis adopts a similar premise to [CSC25] but in a different context; rather than deriving an intermediate representation from visual reference artifacts, this thesis creates an explicit link between software models and user interfaces by generating the latter from the former by means of extracting a layer of interface meta data.

2.7 Object-Oriented User Interface (OOUI) Principles

The object oriented user interface is designed around the objects but not around the actions [Col95]. Its core principle is that the interface is centered on the objects that users interact with, which allows users to directly interact with components instead of interacting with abstract operations [Col95]. Every component has basic operations to create, read, update and delete objects. The navigation structure of the interface reflects the relationships between objects, making sure that users can move naturally between related entities within the system [Col95].

2.8 Human-in-the-Loop (HITL) Approaches

Human-in-the-loop methods integrate human feedback into automated software generation processes [DWLH24, WSK+26]. In UI generation, designers can review generated outputs, provide adjustments or preferences, and iteratively guide the system toward a more suitable result [DWLH24, WSK+26].

2.9 The AI4MDE Framework

AI4MDE is a model-driven engineering framework that generates software prototypes from UML models [vD24, dH24, Wil24]. The framework includes a Django-based generator capable of producing software prototypes from UML class, use-case, and activity models [vD24, dH24, Wil24].
This thesis investigates how UML-derived interaction semantics can be expressed as a structured UI intermediate representation to guide LLM-based interface candidate generation and iterative UI refinement, and at the same time extends the AI4MDE generation pipeline by integrating Large Language Models (LLMs) to generate diverse, context-driven UI candidates guided by object-oriented interface principles and constrained by UML-derived interaction semantics.

2.10 Summary and Research Gap

UI generation techniques can be divided into two categories: those focused on model-driven engineering, or model-to-code, transforming design models into code based on fixed rules, and those focused on LLM-based UI generation, or prompt-to-UI, generating interface code from text based on learned patterns. This thesis deals with a third category of UI generation: the systematic derivation and representation of latent software design models like actor-scoped operations, relations between objects, and workflow structure before generating UI. There are many specific gaps that justify this third direction of UI generation.
Existing model-driven and low-code approaches can reduce manual development effort and generate basic software interfaces from structured models [Dri20, vA22, ACM25, PAA23]. But some limitations persist to generate enterprise-oriented interfaces from UML models.

Existing model-driven UI generation often generates only basic CRUD (Create, Read, Update, Delete) screens or templates [Dri20, vA22], not supporting UML-to-OOUI transformation: where the classifier becomes object-centric views, the association becomes related object navigation and operations can be attached to the object.
Enterprise-oriented applications involve multiple roles with various responsibilities and permissions [Wil24]. However, hardly any existing model-driven approach combines Use Case and Activity diagrams to jointly represent role-specific interfaces and workflow [Wil24].
Existing LLM-based approaches can support flexible UI layout and visualization, but their unconstrained output can be erroneous, hallucinatory and inconsistent with the source code [DMJ+24, CTBV23]. Hence, we need schema-constrained, UML-bound validation and deterministic guardrails [DMJ+24].
Although SpecifyUI[CSC25] demonstrated a way to control LLM generation using a structured intermediate representation based on design images, currently we lack such methods based on software design models.
Many UI generation approaches still use the one-shot generation strategy, without considering the process where a user can choose preferred designs and regenerate until they get satisfying ones [DWLH24, WSK+26].
In this thesis, gaps as such are filled by proposing a combined approach with deterministic UML-to-OOUI transformation, LLM-based UI generation based on predefined schema, verification guardrails and human-in-the-loop candidate regeneration. The implementation extends AI4MDE platform and the following chapter explains the proposed research methodology and system design.

3 Research Approach and Methods

3.1 Methodology

This research follows a Design Science Research methodology, consisting of problem analysis, artifact design, implementation, and evaluation [vD24, dH24].
As shown in Figure 1, OOUI principles are applied during the transformation from UML metadata to Interface Metadata rather than being treated as separate input models [dH24, Wil24]. The resulting Interface Metadata is then rendered by the Django/Jinja2 framework to produce web UI candidates [vD24].
Human-in-the-loop interaction is included so that designers can review generated candidates, select a preferred UI direction, and provide natural language feedback for regeneration. For example, a designer may select one rendered candidate and request a more compact layout, a different colour theme, or a changed navigation style without restarting the UML-to-interface mapping process.
Users can request refinement and candidate regeneration for the web UI. In the interface designer, users can also manually adjust section placement, field ordering, style settings, and layout options for generated UI section components.

3.2 Input Data

Below lists the types of input for the entire system, not for the LLM alone. The input for the LLM is the structured JSON metadata of page and section derived from UML, plus the system predefined prompt, optional prompt and interactive refinement prompt, which is explained in detail in Section 4.4.1.

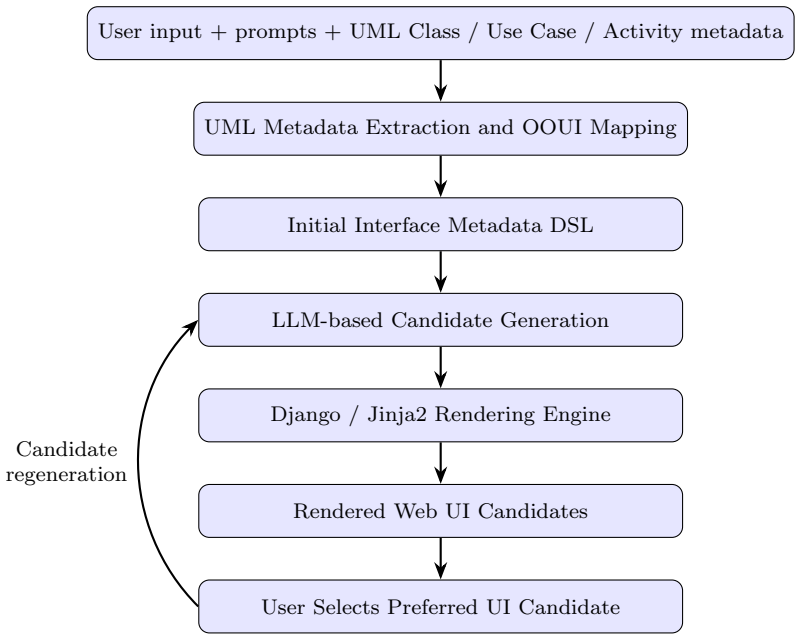

Figure 1: Vertical overview of the proposed AI4MDE pipeline for OOUI-aware UI generation. UML metadata is first mapped to an initial Interface Metadata DSL, after which LLM-based candidate generation produces web UI candidates able to be rendered for user selection and regeneration.

- **UML Class models:** including classes, attributes, and associations in JSON [Dri20, vA22].
- **UML Use Case models:** including actors, use cases, actor-use-case links, and inferred operation scopes in JSON [Wil24].
- **UML Activity models:** including workflows, states, and transitions in JSON [Ver25].
- **Optional prompt text:** instructions or requirements made by the designers to guide the AI to generate specific interface elements [dH24]. For example, a prompt could specify: "Generate 2 (maximum 3) shop interfaces with two actors: buyers and administrators, where buyers can browse and purchase products, and administrators can manage inventory and orders."
- **Iterative refinement prompt text:** an iterative refinement prompt could specify: "put the order details on the right side of the page instead of the left side, or generate the web-shop interface with a blue ocean theme"

3.3 LLM-based UI Generation

A pretrained LLM aids the transformation of UML metadata to Interface Metadata[Zei23]. A hybrid approach is explored, which employs LLM for open design decisions while a schema validator, alongside rule-based constraints ensures correctness and consistency. Non-LLM-driven steps involve UML extraction, page-section mapping, schema normalization, validation and rendering.

3.4 Human-in-the-Loop Iteration

Human-in-the-loop means people adjusting many times in the middle until a satisfactory result is achieved. In the scope of this project, this happens in the process of selecting and regenerating UI candidates. After the 3 UI candidates are rendered by the Django/Jinja2 framework, users can browse the alternatives and select what they like, and send instructive prompts to the chat box in human language to the LLM. Then, the candidate's regeneration process takes the selected candidate as a reference, and makes another group of candidates again. Therefore, users can improve their UI design choices by repeating the above-mentioned steps many times, without establishing the association from UML to web interface from scratch. In our experiment, the number of regenerations is limited. In this way, it is easy to compare the results between participants, and it can also prevent over-reliance on the output of candidates to repeat.

3.5 Verification and Evaluation

This system is validated by conducting a small empirical study involving a user study. Based on prototypes generated from generated code and a subsequent questionnaire, the evaluation avoids performance metrics (task completion time and error rate) as it is a lightweight evaluation study. This study measures the ease of use felt by participants, the compatibility of object-oriented UI, the support of role, the understandability of workflow, and the quality of LLM-powered candidate generation.

The process requires participants to perform the evaluation with one of the available example systems (an e-commerce system, a library system, or a loan application system). They choose a role-based interface and utilize the provided design page of the system platform for aligning UML models into a default interface template, then generate candidate interfaces, apply one preferred candidate to the initial interface and sync it to a live prototype. Lastly they conduct a short interaction with the interface prototype to verify the flow.

At the end of their interaction, the users fill in a standard questionnaire with questions composed on a 5-point Likert scale under various categories including: object-oriented UI navigation support, role segregation, workflow support, general usability, AI candidate diversity, and human-in-the-loop pre-generation. Users are allowed to add optional free-form feedback addressing the elements of the interface that may cause confusion, elements that may prove useful, or general feedback about the interface limitations.

All the questionnaire responses are evaluated descriptively with mean scores across the categories. The open-ended answers will be analyzed with qualitative means to provide general feedback on strengths and weaknesses. The small scale of this study limits the validation on quantitative measures, but it may provide sufficient practical insights for validating whether a system generates interfaces which are readable, role-appropriate, and support the workflow in line with OOUI concepts.

4 System Implementation

This section describes the technical implementation of the proposed AI-driven user interface generation system within the AI4MDE framework. The implementation combines determinis-

tic model-driven engineering techniques with Large Language Model (LLM)-based reasoning to transform UML metadata into interactive web user interfaces.
The system is implemented as a Django-hosted LLM-assisted user interface generation service, deterministic UML mapping modules, and a Django/Jinja-based rendering framework. The architecture supports multi-stage interface generation, multiple candidate variants, schema-constrained validation guardrails, and iterative human-in-the-loop candidate regeneration.[1]

4.1 LLM Interaction Overview

This subsection summarizes the use of the LLM as it is central to the UI generation step. The LLM never receives the raw UML models. Instead, UML Class, Use Case, and Activity metadata is first processed entirely deterministically (Section4.3) into an initial Interface Metadata with information of page and section, and only this already-derived, schema-shaped data is exposed to the LLM. Figure 2 summarises both call types.

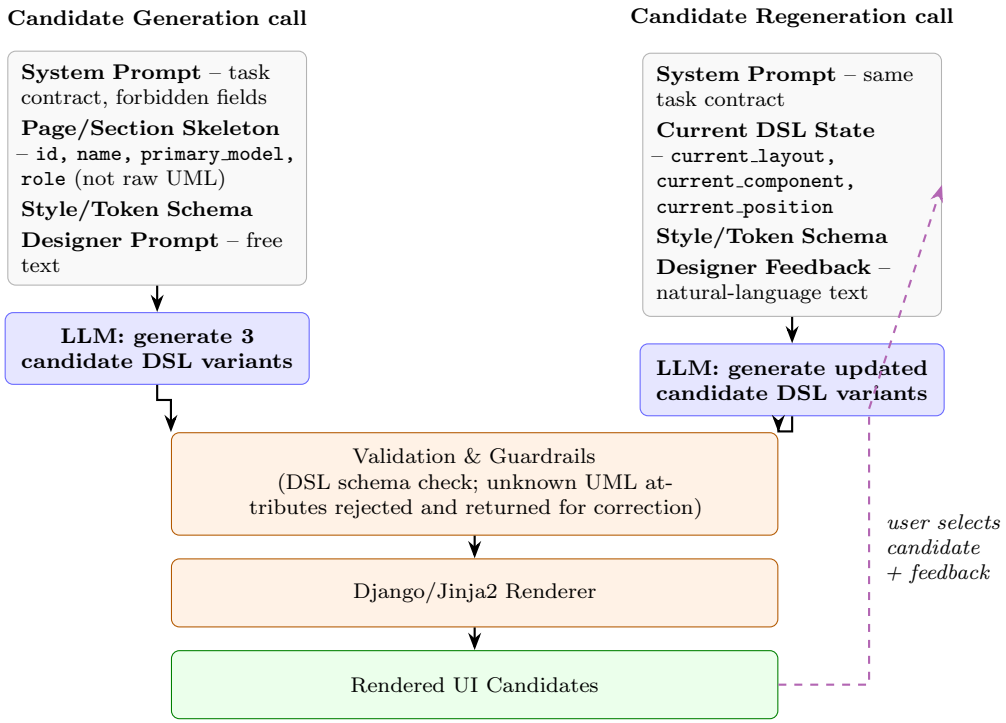

Figure 2: This diagram shows the information sent to LLM at two points when it is called, and shows how to check the output before generating the prototype. Both calls only receive the Interface Metadata DSL item shown above, and never receive the original UML metadata. The dotted arrow indicates the regeneration cycle of human-in-the-loop.

[1]The source code of this implementation is available at: https://github.com/ai4mde/studio/tree/develop-ui-mapper

4.2 Architecture Overview

The proposed interface generation system follows a hybrid architecture that combines deterministic rule based derivation and AI-assisted reasoning. UML Class, Use Case, and Activity models are first transformed into structured metadata representations. These metadata representations are then processed by rule-based planning modules and optional LLM-assisted refinement module to produce Interface Metadata DSL describing the target web user interface. The architecture also supports iterative refinement sessions and integration with the existing AI4MDE pipeline.

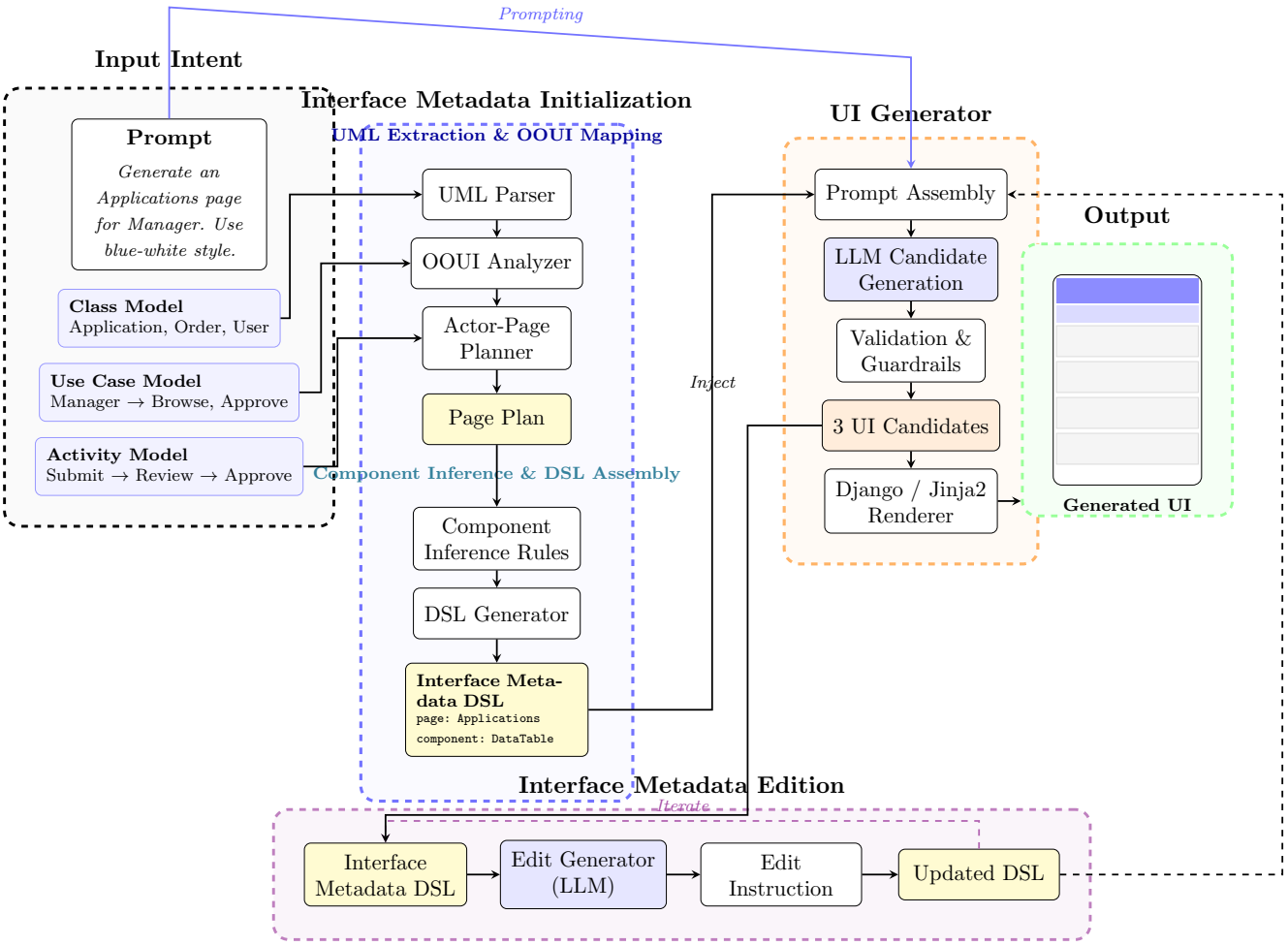

Figure 3: AI-driven UI generation pipeline. Interface Metadata is first initialised from design prompts and UML-derived Page Plan data, not the raw UML metadata itself. This derived Interface Metadata DSL, together with a fixed system prompt and the supported style/token schema, is assembled by Prompt Assembly and sent to the LLM, which returns three candidate DSL variants restricted to layout, component, and styling fields. Validation & Guardrails checks these outputs against the DSL schema and the source UML classifier attributes. For example, a patch introducing an attribute not present on the source UML classifier is rejected and returned for correction – before the Django/Jinja2 Renderer selects a template by component type (e.g. a section with `component: DataTable` is rendered through the DataTable template) to produce the rendered UI candidates. In the Interface Metadata Edition stage, once a designer selects a candidate, the Edit Generator (LLM) receives that candidate's current DSL state together with the designer's natural-language feedback and returns an Edit Instruction that produces an Updated DSL, which re-enters Prompt Assembly, closing the iterative generation or regeneration loop shown by the dashed *Iterate* arrow.

Figure 3 shows the full pipeline. Each role in Figure 3 represents a specific functionality in the code base, not a strict one-to-one correspondence onto source files or modules. The components in the Interface Metadata Initialization stage each serve a distinct role, as described below.
The UML Parser derives class, use-case, and activity information from UML JSON metadata received from the AI4MDE framework. The OOUI Analyzer translates UML classifiers and associations into

page sections related to the object of the class diagram. For example, a section named "Payment Overview" would be assigned to the Payment page, linking to its underlying Payment object. A dedicated inference step, implemented as two functions, leverages the actor-use-case and activity information to define actor-specific page requirements and allocate actor-specific sections across those pages; this is explained later in Section 4.3 .
The Component Inference Rules define declarative YAML rules to translate the context and type of each section's UML signals to a corresponding concrete UI component.
The DSL Generator merges the Page Plan with component types and generates the Interface Metadata DSL, a JSON representation of the interface design, which guides LLM-based generation by structuring and restricting output based on derived actor-scoped, object-oriented model information.

4.3 Metadata Extraction and UML Mapping

Pages and sections are derived from UML diagrams, and this is the processing of mapping. This process uses metadata of class, use case and activity diagrams as input to produce the metadata for the pages, section fields and actor specific operations, determining which page will be created for which actor in the diagram and which section should belong to a page, as summarized in Figure 4. Figure 4 shows how the three types of UML diagrams are combined to derive actor-specific pages, page sections, and concrete UI components. This is a cross-diagram approach, meaning that the pages and sections are derived from multiple types of UML diagrams. Furthermore, by using intermediate JSON metadata to represent diagrams, pages, and sections, the system can produce an actor-scoped, OOUI-guided interface specification that is traceable to the source design models.

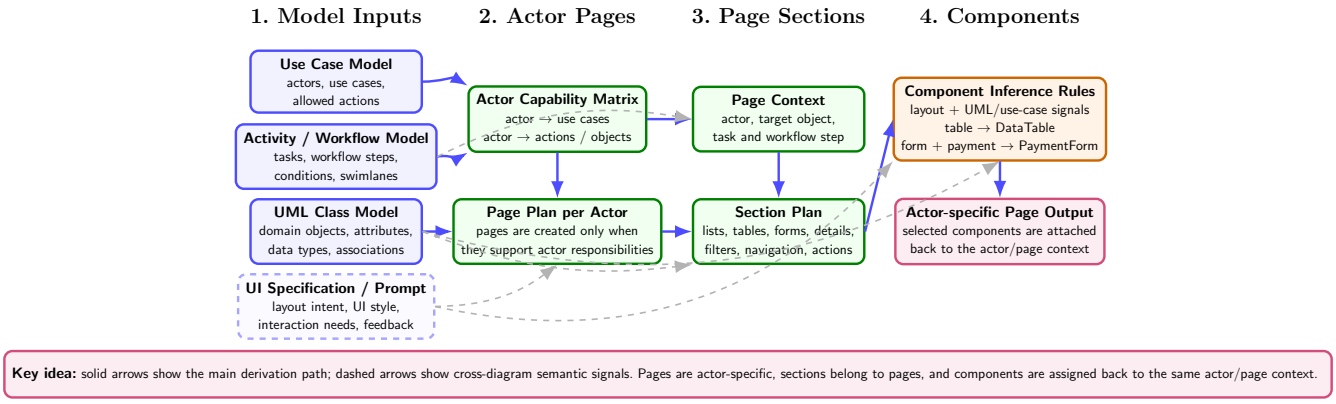

Figure 4: Cross-diagram derivation of actor-specific pages, sections, and components.

Solid arrows in Figure 4 represent the main derivation path: Use Case and Activity models derive the actions each actor can perform, and the Class Model drives the pages. Sections are then created for pages and are fed into Component Inference Rules to produce the final actor-specific page output. Dashed arrows are supporting cross-diagram signals. Every kind of signal plays a role. For example, activity diagrams enrich page context with workflow steps, class diagrams inform section field selection and component inference, and the UI Specification provides style and layout guidance.
The mappings described above are based on rule set predefined in YAML files. YAML files show all mapping knowledge, making the inference logic inspectable and extensible. The rule set is organized

into four layers that execute in sequence.

4.3.1 YAML Transformation Rule Set

Layer 1 Semantic Inference Keyword tables associate UseCase names with page roles, operation types, and CRUD permissions prior to structural mapping. Each use case is assigned a semantic role, such as `collection_workspace`, `detail_workspace`, or `workflow_entry`. Permissions for view, create, update, and delete are determined by the keywords in the name.

```yaml
semantic_inference:
  role_keywords:
    workflow_entry:       [apply, submit, book, schedule, reserve]
    collection_workspace: [select, browse, search, list, track, history]
    detail_workspace:     [detail, "view "]
    object_workspace:     [account, profile, settings]
  permission_keywords:
    create: [add, create, submit]
    update: [manage, update, edit, select, confirm, track]
    delete: [delete, remove, cancel]
```

Listing 1: Layer 1: semantic inference keyword tables (excerpt)

Layer 2 Semantic Model Entity intent rules assign each class to a specific interaction intent profile. Primary intents (Lookup, Configuration, CRUD) are determined by the first matching rule. Secondary intents (Search, Hierarchy, StateTransition) are added as applicable. Activity diagrams are assigned workflow intent profiles (Decision, Sequential) by matching action node names to workflow patterns.

```yaml
semantic_model:
  intent_rules:
    - primary: [Lookup]
      when: { max_operations: 0, max_attributes: 6 }
    - primary: [Configuration]
      when: { max_operations: 0, name_matches: "(setting|permission|config)" }
    - add: [Search]
      when: { primary_is: CRUD, min_attributes: 4 }
    - add: [Hierarchy]
      when: { computed: from_composition }
    - add: [StateTransition]
      when: { attribute_name_matches: "^(status|state|phase|stage)$" }
  workflow_patterns:
    - intents: [Workflow, Decision, StateTransition]
      match:
        requires_decision: true
        action_name_matches: ["(approve|accept|confirm)", "(reject|decline|deny)"]
    - intents: [Workflow, Sequential]
      match:
        action_name_matches: ["(fill|enter|submit|provide|upload)"]
```

Listing 2: Layer 2: entity intent rules and workflow patterns (excerpt)

Layer 3 UML Mapping Rules Five main rule groups connect UML to Interface Metadata Domain-Specific Language (DSL) constructs. These groups use declarative expressions: (1) mapping UseCase Actor to Interface, (2) mapping UseCase to Page, (3) mapping Class to Data Section and Association to Related Section, (4) mapping Activity Action Node to Activity Page and Action Section, and (5) a cross-diagram post-processing step that links class-derived sections to actor assignments from activity diagrams. In the fifth group, sections derived from class diagram are matched to the appropriate actor interfaces using swim-lane(an activity action node's swim lane points to an actor classifier named 'Customer'; if that classifier's type is actor, the action is assigned to the Customer's interface) information from the activity diagram. This ensures each domain object section appears under the correct actor's interface, rather than being assigned to all interfaces. For example, `uc.cls.name` is resolved to the string `"Browse Products"`, which is the name of the current class being iterated, and then this string is fed to the `infer_role` function, which searches in `semantic_inference.role_keywords` to find a keyword matching `"browse"`, to check if `"browse products"` contains a keyword, and the corresponding role (such as `collection_workspace`) will be returned if a keyword matched. The same goes for `infer_permissions` which uses `permission_keywords` instead of `role_keywords`.

```yaml
rules:
  - id: usecase_node_to_page           # Group 2: use case diagram -> Page
    from: { diagram_type: usecase, ... }
    mapping:
      role:        "$uc.cls.name | infer_role"       # from use case name keywords
      permissions: "$uc.cls.name | infer_permissions"

  - id: association_to_related_section  # Group 3: class diagram -> Related Section
    from: { diagram_type: classes, edge_type: association,
            source_multiplicity: ["*","0..*"], target_multiplicity: ["1","0..1"] }
    mapping:
      layout:     "table"
      related_to: "$source_section.id"   # links to parent class section

  - id: assign_sections_to_interfaces   # Group 5: cross-diagram synthesis
    strategy: post_assign_sections
    # For each section in the class_section registry (built from class diagram):
    #   if the section's class appears in activity_actor_class registry
    #     (built from activity diagram swimlanes) → assign to that actor's interface.
    #   else → assign to all interfaces (fallback).
    # Result: class-derived sections are scoped by activity-diagram actor context.
```

Listing 3: Layer 3: UML mapping rules including cross-diagram synthesis in Group 5 (excerpt)

Layer 4 Section Composition Section composition patterns integrate conditions from all three diagrams in a single match. The use case diagram (Layer 1) sets `page_roles`, class diagram entity classification (Layer 2) provides `intents_any`, and activity diagram analysis (Layer 2) supplies fields such as `workflow_intents_any` and `activity_terms_any`. A pattern is triggered only when all conditions are met, ensuring that section composition decisions reflect semantics from multiple diagrams instead of from a single diagram. Seventeen named patterns address both activity workflow pages and UseCase workspace pages.

```yaml
section_composition:
  patterns:
    - pattern: ApprovalStep
      page_roles: [activity_action]          # activity diagram: page type
      workflow_intents_any: [Decision]       # activity diagram: branch present
      activity_terms_any: [assess|review|decide|final.decision]  # activity node name
      sections:
        - { role: object_summary,  component: SummaryPanel }
        - { role: workflow_action, component: ApprovalActionPanel }

    - pattern: MasterDetail
      page_roles: [collection_workspace]     # use case diagram: page role
      intents_any: [Search, StateTransition] # class diagram: entity intent profile
      sections:
        - { role: filter,            component: FilterPanel }
        - { role: object_collection, component: $collection_component }

    - pattern: MasterDetail
      page_roles: [detail_workspace]         # use case diagram: page role
      intents_any: [Hierarchy]               # class diagram: owns 1:N children
      sections:
        - { role: object_detail,    component: $detail_component }
        - { role: child_collection, component: RelatedObjectList,
            for_each: compositions_1n }
```

Listing 4: Layer 4: section composition patterns combining conditions from multiple UML diagrams (excerpt)

The four-layer structure shows how three different diagram types of semantics are combined into an interface specification. Layer 1 infers CRUD actions by keywords of classes in each class diagram and Layer 2 derives entity intent intermediate rules by semantic information. Layer 3 maps these elements to the components of domain-specific language (DSL) and integrates the first diagnostic program (Group 5). Layer 4 combines the signals obtained by using the pattern condition to determine the structure of the section suitable for each page. Each result of assigning a Page, Section, or Component produced will trace back to the source of the specific rule describing the derivation.

4.4 UI Candidate Generation and Regeneration

In the implementation, candidate generation and regeneration are handled differently. Candidate generation creates three visually distinct variants from the same UML-derived page and section structure. Regeneration starts from a selected candidate and incorporates user feedback. Listing 5 shows the kind of structural context passed to the model.

```
Pages: [{"id": "...", "name": "...", "type": "..."}, ...]
Sections: [{"id": "...", "name": "...", "primary_model": "...",
            "role": "..."}, ...]
```

Listing 5: Page and section skeleton passed to the candidate generation prompt

The pages and sections derived from UML mapping are passed to the LLM so that layout and style can be varied without removing the underlying actor, object, workflow, and data-binding information.

4.4.1 Prompt Design

Prompt design is the predefined set of prompts in the system that guide the LLM to fill in fields of the Interface Metadata in the fixed AI4MDE pipeline. Both predefined prompts in the system and additional prompts made by users serve as rules or requirements for the LLM to tweak UI interfaces: they specify what the LLM should do and target the JSON fields for the styles and layouts that can be edited. This is important for the research goal because the system does not let the LLM make interface code at will, due to the limited scope that can only be edited by prompts. Instead, it will guide LLM to make design judgments in the interface metadata created from the model metadata(UML).

```
You are a UI designer creating 3 structurally and visually distinct
interface layout candidates. Follow the schema and role rules exactly.

You output layout + style decisions for an interface. DO NOT change:
id, name, primary_model, class, attributes, operations, role,
behavior, data_source, query, field_layout.
```

Listing 6: System prompt for candidate generation

This prompts will display the corresponding interface DSL fields in the model, containing page, section, layout, component, styling and token fields. LLM is not going to edit the predetermined fields in UML mapping. This is the case with attributes, operations and data source. This is soft prompt to add a security layer. Soft prompts mean an additional security layer, as even if there is no such prompt, the barrier check will eliminate the restricted fields from the output of the LLM. This can prevent arbitrary field values or CSS from appearing in the model, and make the generated output conform to verification and drawing. On the additional guardrail, the data-bound section only uses the classifier attributes from UML to distinguish the real model image field from the static image URL, and the user obviously requires greater changes. Therefore, the design of the prompt cannot replace the verification of fixed methods. Instead, it restricts the LLM's behavior. In this way, the idea of the model will lead to the update of the structured DSL, and the downstream system can check, modify and render it. In order to make the three candidates diverse, the user's prompt has clear diversity constraints. As shown in Listing 7.

```
Generate exactly 3 structurally and visually distinct candidates.
Each candidate MUST follow user requirement.
Each candidate MUST also differ in axes that user did not specify.
Every page must have navigation unless user requires otherwise.
When the designer prompt does not specify width, vary page widths
across the 3 candidates: contained, wide, and full.
Respect the designer prompt and choose colors as a UI designer.
```

Listing 7: Diversity constraints injected into the candidate generation prompt

For regenerating UI, the global layout and style settings are inherited from the selected candidate in the first iteration. The design attributes of this base candidate including font family, accent colour, background colour, border radius, button style, and card hover effect, are injected into the prompt so that the regenerated candidates use the selected candidate as their visual starting point (see Appendix A for the full template).

The regeneration diversity prompt follows the same principle as the initial candidate diversity prompt. It asks each regenerated candidate to differ along design axes that are not fixed by the user's feedback, such as typography, density, border radius, button style, card hover effect, page width, and navigation placement, while preserving the base candidate's section roles and data bindings. The full regeneration diversity prompt is provided in Appendix A.

Listing 8 shows the section-level skeleton used during generation and regeneration. Unlike the global styling context described above, this skeleton exposes the current visual state of each section in the page, so that the LLM can decide which layout or style choices should be preserved and which can be changed.

```
// Generation: semantic context only
{"id": "...", "name": "...", "primary_model": "...", "role": "..."}

// Regeneration: includes current visual state
{"id": "...", "name": "...", "primary_model": "...", "role": "...",
 "current_layout": "...", "current_component": "...",
 "current_position": "..."}
```

Listing 8: Section skeleton passed to generation (top) and regeneration (bottom). Regeneration includes current visual state to anchor the LLM to the selected candidate.

Listing 9 shows how designer feedback from the frontend chat box is inserted into the regeneration prompt before it is sent to the LLM.

```
DESIGNER REQUIREMENTS: <user input or "(explore visual variations)">
```

Listing 9: Designer requirements field in the regeneration prompt

Listing 10 shows a condensed excerpt of the style schema used in the prompt contract. The schema limits the LLM to supported layout, and styling fields instead of allowing arbitrary CSS. High-level

fields control component presentation and page layout, while fine-grained tokens provide exact colour overrides for specific regions such as headers, cards, buttons, inputs, and navigation.

```
style:
  color: "accent" | "blue" | "green" | "purple" | "rose"
  density: "compact" | "normal" | "spacious"
  columns: "1" | "2" | "3" | "4"
  display_mode: "grid" | "carousel" | "banner"
  card_style: "default" | "product" | "compact"
  image_ratio: "wide" | "16:9" | "1:1"
  shadow: "none" | "sm" | "md" | "lg"

page:
  layout.value: "vertical" | "horizontal"
  layout.main_width: "contained" | "wide" | "full"
  gap.value: "compact" | "normal" | "spacious"

global styling:
  fontFamily: "inter" | "roboto" | "poppins" | "geist"
  textSize: "xs" | "sm" | "md" | "lg" | "xl"
  accentColor: "#hex"
  backgroundColor: "#hex"
  textColor: "#hex"
  radius: 0 | 4 | 8 | 16 | 24
  buttonStyle: "solid" | "outline" | "ghost"

fine-grained tokens:
  region.header.bg_hex: "#hex"
  region.footer.bg_hex: "#hex"
  component.card.bg_hex: "#hex"
  button.primary.bg_hex: "#hex"
  input.bg_hex: "#hex"
  input.border_focus_hex: "#hex"
  nav.bg_hex: "#hex"
  text.muted.hex: "#hex"
```

Listing 10: Condensed excerpt of the style and token schema exposed to the LLM

4.4.2 Human-in-the-Loop for Candidate Regeneration

Candidate generation and regeneration are different functionalities. Candidate generation spawns three unique visual iterations of the same set of pages and section designs based on user prompts, and a readily available default UML-derived page and section metadata, marking the first iteration of human in the loop for generating UI. On the other hand, regeneration is based on a chosen candidate and blends iterative user prompts in the regeneration process. Figure 5 and Figure 6 illustrate this.

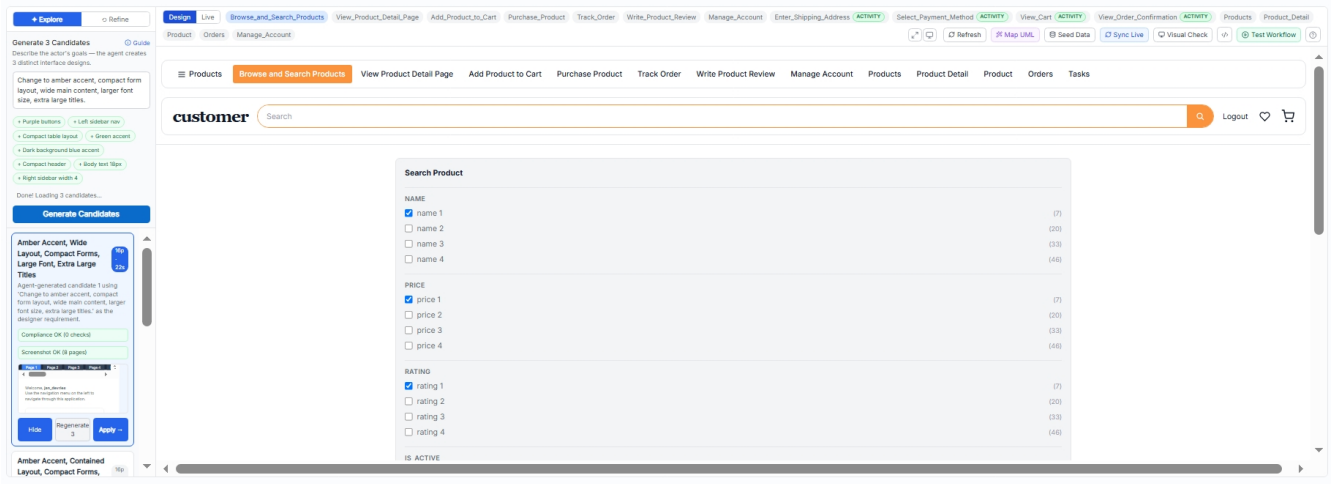

Figure 5: Candidate generation: The designer enters the natural language prompt to the chat box in the left panel and the system will make three interface candidates with different appearances, whiling adhering to user's prompt as much as possible.

After reviewing generated interface candidates, designers can select a candidate they would like to have and give iterative prompt such as:

"Make the dashboard more compact"

"Use a blue colour theme"

"Move order details to the right side of the page"

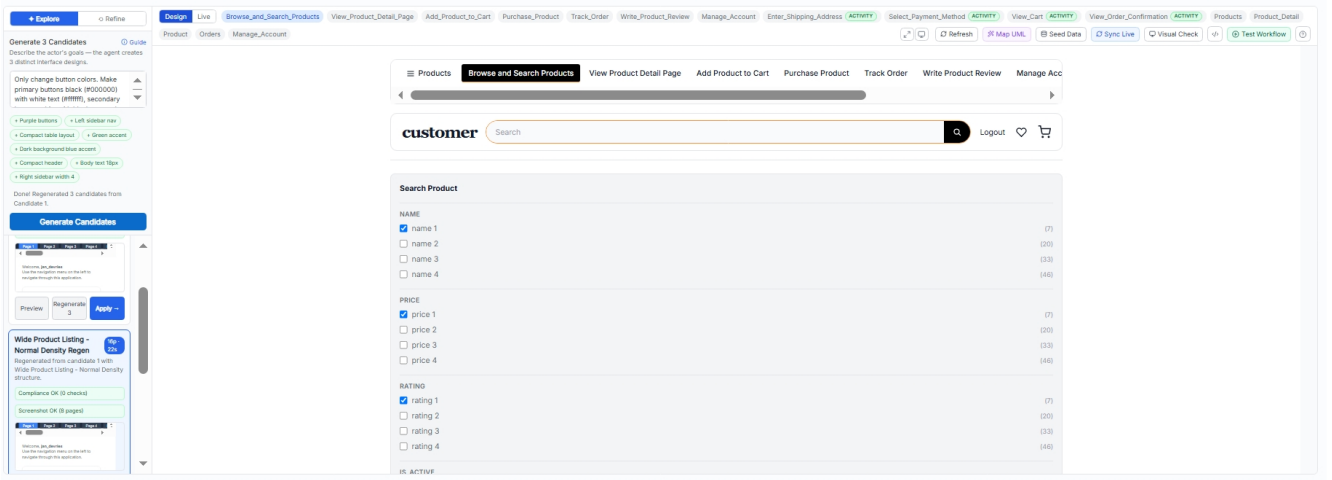

Figure 6: Candidate regeneration: the designer will "only change button colors. Make primary buttons black (#00000) with white text (#fffff)". The system will not process UML mapping from scratch, but recreate three new variants from the selected DSL. The real-time preview will immediately display the new style tokens.

Regeneration is based on the styles and layout items in the selected candidate interface DSL. The output uses the same DSL structure, and the complete Interface Metadata will not be recreated from the beginning.

This process has a direct relationship with human-in-the-loop hypothesis: the structure of the pages and parts derived from UML model can be preserved, and the styling will align under the guidance of the designer's ideas through prompt iteration. Refinement adjusts JSON fields of selected DSL candidates within a limited range. Instead of starting the UML-to-interface mapping process from scratch, the pages, sections, layout configuration, styling and design tokens currently are part of the selected candidate, and output of regeneration will generate a group of new candidates written in the same interface DSL schema, as shown in Figure 6. Figure 7 shows what type of natural language feedback corresponds to which part of Interface Metadata JSON: style adjustment will update style and tokens layout and feedback from component will change attributes of a section, and page structure requirements will change Page Specifications.

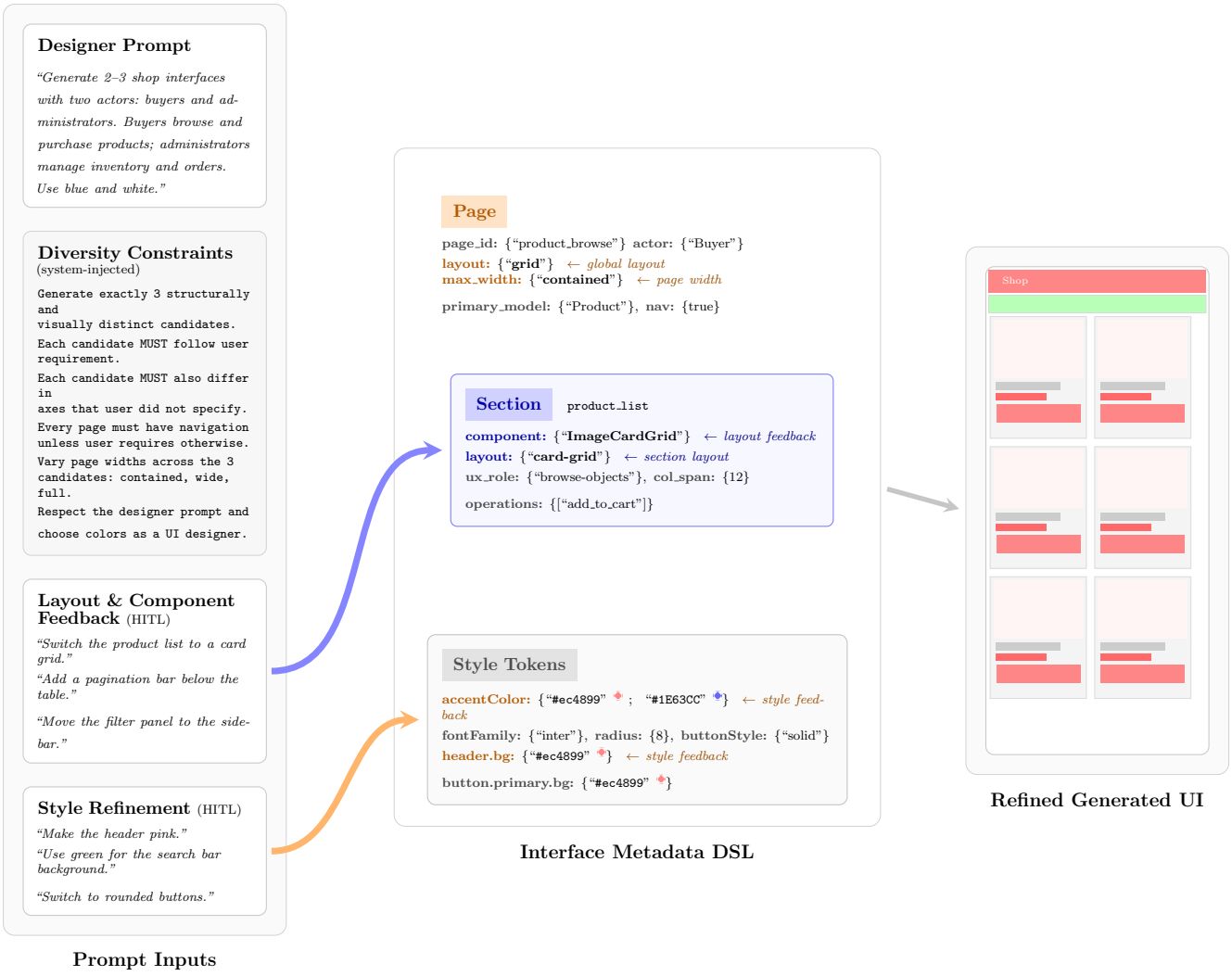

Figure 7: Human-in-the-loop feeds back the region corresponding to Interface Metadata DSL. Style adjustment (color, font) will update Style Tokens, layout and component feedback will change Section Attributes, and page structure requirements will change Page Specifications. The updated DSL will regenerate improved UI candidates.

4.5 Validation and Guardrails

A challenge in large language model (LLM)-based software generation is ensuring consistency between the generated output and the source Unified Modeling Language (UML) models, as LLMs can generate unsupported fields, invalid component configurations, or hallucinated model attributes [DMJ+24, CTBV23]. As a result, LLM outputs undergo deterministic validation before rendering in the implementation.

The first guardrail is the Interface Metadata DSL schema. LLM cannot make arbitrary HTML, CSS or frontend code. Instead, only page fields, section fields, layout identifiers, component names, operations, styling fields and design tokens can be used. The second guardrail is UML-bound attribute validation. The names of the generated fields will be matched according to the corresponding UML classifier properties before being accepted. Patches referencing unknown

attributes are rejected and returned to the LLM for correction.
In addition, the range of appearance changes is limited by the definitions of style and token fields that can be used. The LLM outputs these changes to a corresponding set of style fields and token fields in the DSL. The renderer then converts it to UI. Finally, there are other steps that help make better use of the DSL output produced by the LLM. For example, if the LLM cannot be reached or returns an incorrect answer, an interface plan based on fixed rules is used instead to fill in the incorrect fields in JSON. To sum up, the system can not only use the reasoning ability of LLM, but also follow the correct steps in the authorization, standardization and drawing of control.

4.6 Rendering and Prototype Generation

The renderer does the last stage of pipeline work, converting the final Interface Metadata into a running prototype using a Jinja template for end-users to use.
The Interface Metadata is validated and then submitted to a Django-based rendering engine that generates live web prototype pages and also the view for the in-browser previews. Both candidate previews and the prototype page rendered during production share a common Jinja2 template level. Because all fields from the interface DSL, the component mapping, the style definitions, and token values are treated the same in both the template previews and the actual running prototypes, the appearance during a candidate-selection visit is identical to the actual running product. This is our approach for bridging one of the major seams that exist between many generative UI prototypes and the ultimately rendered outcome.
Each page is defined as an ordered list of section identifiers. Each section is then resolvable to a component template via layout and component properties. Data-bound sections link classifier attributes to the corresponding columns in tables, form elements, fields in cards, or list or detail items. Chrome sections define structural elements like header and footer, nav-bar, search bar and actions-group.

4.7 Interface Metadata DSL

This resulting interface is captured in a JSON-based Domain Specific Language (DSL), acting as a model in-between for the UML model and context, and the interface prototype[vD24]. This DSL represents the structure and layout, navigation, style tokens, and the components which make up user interface prototypes. It features high-level keys of pages, sections, styling preferences, design tokens, categories, and variants.

4.7.1 Page and Section Schema

Every resulting interface candidate can mainly be decomposed in pages and sections structure. Pages describe the application structure, general page style and layout. Sections belong to pages and section schema contains information such as its layout, the display type, e.g form or card. Which attributes are displayed and which operations are supported are also described by section, as well as some additional behavioural information.
This segregation makes changes possible without section duplication and allows for restructuring of the pages whereas sections may be updated individually on the basis of feedback and validation,

and this abstraction enables the downstream AI4MDE pipeline to convert the DSL into tangible interface artifacts that may include for instance HTML prototypes or front-end components
Listing 11 shows a simplified example of the generated DSL. The example represents an `Applications` page generated from the UML class `Application`. The page references a table section, while the section stores the actual model binding, attributes, operations, and style configuration.

```
{
  "pages": [
    {
      "id":            "applications",
      "name":          "Applications",
      "primary_model": "Application",
      "type":          { "value": "normal" },
      "nav":           true,
      "layout":        { "value": "vertical", "main_width": "contained" },
      "sections":      [{ "value": "applications_table" }]
    }
  ],
  "sections": [
    {
      "id":             "applications_table",
      "page_id":        "applications",
      "layout":         "table",
      "component":      "DataTable",
      "primary_model":  "Application",
      "visible_fields": ["name", "status", "submitted_at", "applicant"],
      "field_layout": {
        "columns": ["name", "status", "submitted_at", "applicant"],
        "hidden":  ["id", "application_id"]
      },
      "operations": ["read", "update"],
      "style":      { "color": "accent", "density": "compact" },
      "col_span":   12
    }
  ]
}
```

Listing 11: Simplified page and section objects in the generated interface DSL

4.7.2 Design System Tokens

The visual styling in the DSL is encapsulated in the form of a semantic token to produce detailed design styles for pages and sections without altering an object binding, navigation logic or flow of events.
There are two complementary layers of styling involved in the DSL. The first is a `styling` object, that contains high-level decisions that should drive the high-level styling, such as: font choice, the main accent color, card hover behaviour, the border radius of elements, button styling, and the

desired width of the page. The `styling` object serves as a concise overview of how the user wants their pages styled and can also be read by an LLM to refine further styling decisions.
The second layer is the `tokens` object that contains explicit rendering values. These are precise hexadecimal values and other component-level defaults including page `backgroundColor`, `header backgroundColor`, `footerBackgroundColor`, `navigation`, `cardSurface`, and styling for `buttons` and `input` elements as well as table headers and muted text, that the rendering layer can then render out as Django/Jinja2 variables into CSS variables, and even Tailwind-ready classes. Examples of high-level styling fields include:

```
{
  "styling": {
    "accentColor": "#ec4899",
    "backgroundColor": "#ffffff",
    "textColor": "#111827",
    "radius": 8,
    "fontFamily": "inter",
    "buttonStyle": "solid",
    "cardHover": "lift"
  },
  "tokens": {
    "region.header.bg_hex": "#2563eb",
    "region.header.text_hex": "#ffffff",
    "input.bg_hex": "#dcfce7",
    "input.border_focus_hex": "#16a34a",
    "button.primary.bg_hex": "#ec4899",
    "button.primary.text_hex": "#ffffff",
    "component.card.bg_hex": "#ffffff",
    "text.muted.hex": "#6b7280"
  }
}
```

Listing 12: Example of high-level styling fields and fine-grained rendering tokens

The fact that the approach is token-based also allows better control over the generated candidates as tokens capture more design styles than styles: the same OOUI page-section graph can be used to generate three visually different design candidates for the interface. And during a human-in-the-loop refinement, for example, a user can issue a request to “make the header blue” or “use pink primary buttons” just by modifying the concerned design tokens instead of regenerating underlying pages and sections.
Creating structured design fields with constraint serves as a safety mechanism against creating garbage CSS. The token layer constrains visual output to specific token fields instead of letting LLM generate arbitrarily any set of CSS code. This will limit incorrect styling, will possibly help with preview–prototype fidelity, and gives the renderer ability to present consistent changes between produced views and created prototype page.

4.8 Summary

This section details how the AI-driven UI generation system works in the ai4mde framework. From the architecture point of view, this mechanism first uses a rule-based approach to separate pages and assign section components to pages based on UML metadata. Then, the architecture support users to feed prompts to LLM to tweak the fields of the Interface Metadata in terms of visualization and layout. In addition, the validation and guardrails checking checks if the LLM output is usable, rendering and prototype generation section describes how the DSL is converted to running prototype ready for use. The next section will verify the performance of the proposed method by using usability testing, evaluating whether it meets the rules of object-oriented user interface and to test whether participants can navigate through enterprise workflow scenarios.

5 Evaluation

5.1 Evaluation Setup

Because LLM generated artefacts can differ significantly in terms of their quality and consistency it is essential to perform some form of empirical evaluation to test usability and semantic accuracy of our approach. Therefore in addition to measuring general usability, we also evaluate role isolation, workflow support, object centred exploration, diversity of candidates and quality of refinements in the study presented here. A study was carried out with 9 participants: 7 students, 1 software developer, and 1 participant from a different area. In terms of past experience with low/no-code platforms, 2 had never worked with any, 5 worked once or twice, and 2 used them frequently. 2 participants had no UI design experience, 5 knew some principles without having designed anything, and 2 had created interfaces for private use/school projects. We asked participants to explore some case studies (Hospital, Library, Ecommerce, and Loan Application) and play assigned user roles in the systems. We assessed general usability, role separation, workflow support, object centered exploration, variety of AI candidates, quality of refinement in human-in-the-loop, and estimated utility.

5.2 General Usability

Figure 8 and Table 1 present the general usability results. The box plot shows the spread of participant responses, while the table reports the corresponding mean and standard deviation values for each questionnaire item.

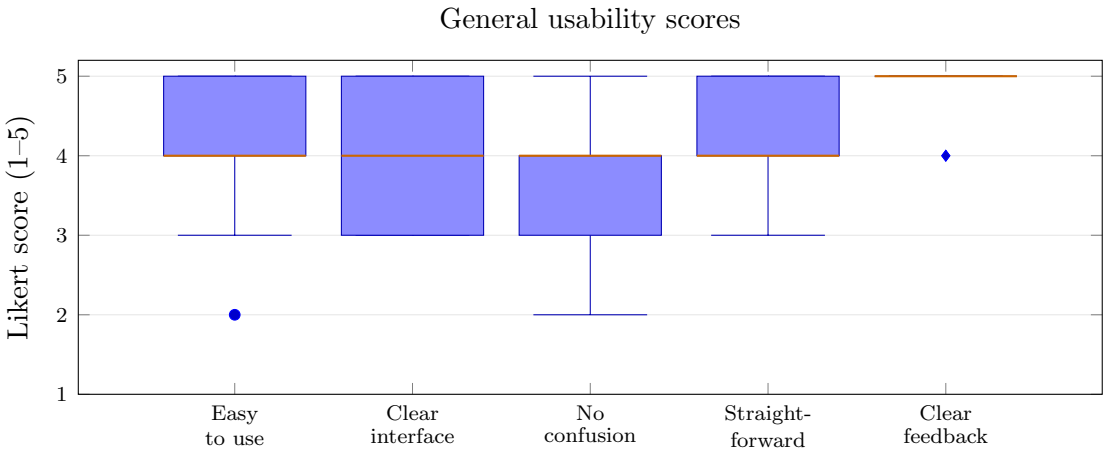

Figure 8: Distribution of general usability scores across participants

Item	Mean	SD
Q4. The system was easy to use	4.00	1.00
Q5. The interface was clear and intuitive	4.11	0.93
Q6. I could complete tasks without confusion	3.56	1.13
Q7. The steps to generate a prototype were straightforward	4.22	0.67
Q8. The feedback from the system was clear	4.78	0.44

Table 1: General usability scores (n=9, scale 1–5)

The means and standard deviations of the five general usability questions are in Table 1. These five items used a five-point Likert scale, where 1 = 'Strongly disagree' and 5 = 'Strongly agree'. Overall, the participants responded positively regarding usability.
The system feedback was the item receiving the highest score on average (M = 4.78, SD = 0.44), implying that the loading indicators and status messages were typically considered clear.
Participants also thought positively of the prototype generation steps (M = 4.22, SD = 0.67) and interface (M = 4.11, SD = 0.93). The question about task completion without confusion has the lowest mean score (M = 3.56, SD = 1.13), which corresponds to our earlier analysis. Note that for this question, the SD was relatively high compared to other questions, implying a wide variation of experience in the group of users. Therefore some participants may have experienced clear task execution without doubt while others had difficulty in using the interface smoothly.
This indicates the necessity for more in-system guidance.
Figure 8 shows a box plot of the participants' answers for the general usability questions. The feedback clarity shows more results with values close to 5, whilst the task completion without confusion shows a broader range of values, especially close to 1, supporting that the feedback was quite good, but the task guidance needed to be clearer.

5.3 Detailed Evaluation Dimensions

The overall detailed comparison results were presented through Fig. 9 & Table 2, where figure shows variation over participants, and the mean and stddev were recorded in the table.

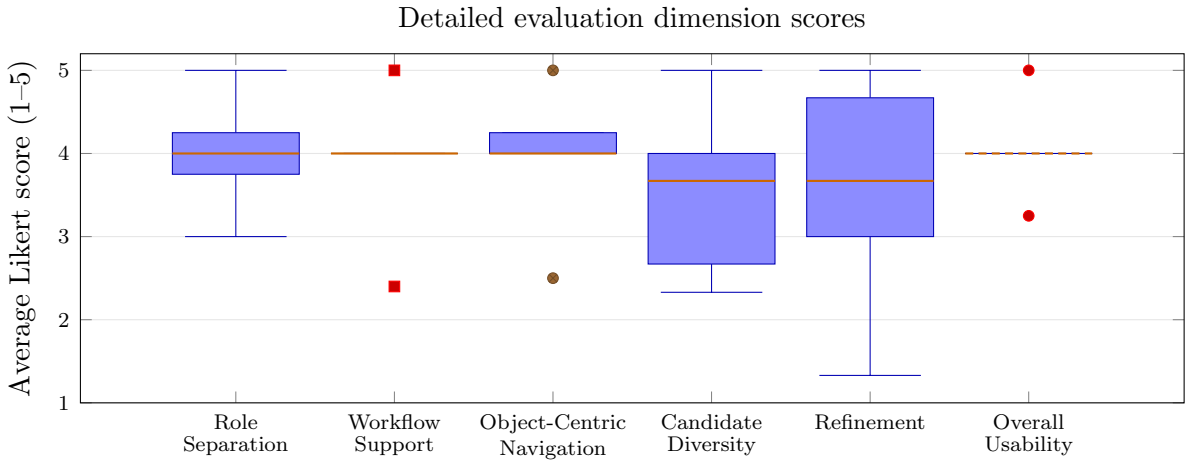

Figure 9: Distribution of detailed evaluation dimension scores across participants

Dimension	Mean	SD
Role Separation	4.00	0.73
Workflow Support	3.88	0.93
Object-Centric Navigation	3.95	0.91
AI Candidate Diversity	3.53	1.07
Human-in-the-Loop Refinement	3.53	1.46
Overall Usability	4.05	0.62

Table 2: Detailed evaluation dimensions (n=5, scale 1–5)

A total of 5 participants fully completed the detailed questionnaire (e.g., for all domains, roles and questions), comprising the dimensions: role separation, workflow support, object-centric navigation, diversity of AI candidates, human-in-the-loop refinements, and general usability. The average scores and standard deviations for each dimension are shown in table 2. It shows that the overall usability ($M = 4.05$, $SD = 0.62$) received the highest score among all, followed by the other four domains where all the mean scores are above 3.6 and only slightly different, except for object-centric navigation ($M = 3.95$, $SD = 0.91$), role separation ($M = 4.00$, $SD = 0.73$).

For the detailed result on workflow support, average score of this dimension is $M = 3.88$, $SD = 0.93$, with item-level results showing the majority of participants agree that the interface guides them to follow a step-by-step procedure logically (WF1: 4.2), but with higher difficulties in following a multi-step procedure (WF5: 3.4), and lack of enforcement to avoid workflow related errors (WF4: 3.6).

This suggests that our generated interfaces have capability of describing workflow structure, but may need more efforts to guide users in complex multi-step scenarios.

AI Candidate Diversity and Human-in-the-loop Refinement had the lowest mean score among the detailed dimensions (M = 3.53 for both). Candidate diversity exhibited a reasonably high spread of scores (SD = 1.07), implying that participants had varying perceptions on how diverse or useful the suggested alternative candidates were. Human-in-the-loop refinement has a broad range (SD = 1.46), showing varied use of the refinement functionality by participants.
The refinement has been functional, but not perfectly accurate and reliable for use.
Figure 9 depicts the scores of each participant for each of the detailed evaluation dimensions. The boxplot indicates that refinement and candidate diversity spread across a wide range compared to overall usability and role separation, which supports the interpretation that participant perceptions of usability and the general role separation of the interfaces were more uniform across participants compared to candidate diversity and the use of the refinement tool.

5.4 Perceived Utility

Users also considered the tool to save time greatly (Q13: 4.4) and indicated they would gladly utilize it in a future project or study (Q14: 4.1), indicating possible perceived utility beyond the evaluation setting.

5.5 Qualitative Feedback

Participants highlighted the speed of prototype delivery and the ability to quickly explore layout options as the most valued aspects of the system. One participant noted that the tool removes the need to manually research and write UI code, while another appreciated the direct path from UI design to a runnable software prototype.
Participants provided three areas of suggestion related to improvements: the need for better inline help or explanations (reported by two users), the need for better, more robust data-binding in the preview mode and prototyped output, and the need for a stronger visual hierarchy in the prototypes of the interactive controls. Those concerns are, not surprisingly, related to the poorer scores we see for human-in-the-loop correction and task completion.

6 Conclusion

6.1 Summary of Contributions

This thesis makes three contributions. First, it develops formal inference rules to translate UML class, use case, and activity diagrams into a hierarchical Page–Section–Component interface specification according to the OOUI principles, in which each produced Page, Section, and Component can be traced to one specific source UML construct: an actor role, class attribute, association or activity node, thereby connecting specific design choices with interface structure in a clearly traceable way.
Second, it defines a notation-agnostic DSL called the Interface Metadata DSL, which captures the actor's scope, domain object structure and activity flow as a form of structured input to LLM-driven UI generation.
Third, it presents a live interaction pipeline to interact with interface model generated from UML and reconstruct UI: users provide natural language descriptions for individual Interface Metadata

JSON fields, the generation pipeline regenerates all interface artifacts, and the users are shown the updated prototype immediately.
Together, the contributions position my work in the space between traditional model-driven UI code generation (using a formal model as input but generating code with specific tooling), and prompt-driven UI generation (using natural language to describe the UI and generating it with an LLM); structured design semantics guide and constrain the LLM output while NL interaction makes the refinement approachable. This connects with recent studies on specification-driven UI generation (e.g., [CSC25]), where an intermediary, structured format proves more effective in controlling and yielding an accurate result compared to relying purely on prompt engineering. This work takes this approach to the next level by using a formal modeling language (UML), as opposed to visual artifacts (such as mock-ups or Wire frames), as the source format.
To conclude, recent work has found LLMs being utilized for a broad range of software engineering concerns, including code generation, support of modeling tasks, as well as generating user interface elements [HZL+24, CTBV23, WSL+24]. Additionally there is a body of recent work on how to incorporate LLMs into existing Model Driven Engineering (MDE) work-flows [NMR24, RBW+25]. The context is the present work, that is examining how explicit context - organized as an interaction semantics - in MDE can be used to constrain LLM-guided generation of UIs for software prototype.

Academic contributions. From a research perspective, the thesis is positioned at the interface of model-driven engineering and LLM-assisted software engineering. It presents a new lightweight semantic structuring layer that is positioned between open-ended, prompt-driven generation and a completely formal, DSL-based, model-driven engineering. This is justified by the model-driven engineering concept that software artifacts can be synthesized from explicit models and intermediate representations [Dri20, vA22, Fow10] and that it has recently been demonstrated that LLMs can be incorporated into model-driven software engineering workflows [NMR24, RBW+25].
It is neither a formal constraint based low code platform nor a fully LLM-based solution. Instead, it arranges UML/domain models, use-case/workflow models, UI specs, and feedback into a structured generation context which operates as constraints for the LLM to further revise UI.
Second, our thesis models and makes the chain of actor-page-section-component generation able to be inspected explicitly. The actor picks the pages pertinent to its responsibilities; pages are structured in sections as a function of actor, object and workflow context and tasks; sections are mapped into concrete UI elements before the resultant elements are returned and composed on the actor-specific pages. With this explicit exposure of model-to-UI generation reasoning process, the entire logic from the diagram to the actor-specific pages, the page sections, the interaction driven by the workflow and the final generated UI can be traced by the user.
This part of our contributions adheres to the general principle in MDE and trace-ability, stating that generated artifacts must preserve a traceability relationship with their source models and requirements [RJ01, ARNRSG06].
Moreover, our work builds upon the OOUI initiative that is to structure an interface on the basis of domain object and its relation-ships [Col95, Ver25].
Overall, this thesis contributes an exploratory AI4MDE-oriented UI prototype generation method and prototype system. By explicitly organizing domain objects, actors, workflows, page contexts, and component mapping rules, the approach makes LLM-based UI generation more structured, traceable, and aligned with application semantics while remaining lightweight enough for early-stage software prototyping.

6.2 Answers to Research Hypotheses

6.2.1 H1: OOUI Principle Reflection and Traceability

The proposed strategy targets H1 with a formal inference rule set that transforms a UML model into a hierarchal Page-Section-Component based interface specification. The resulting elements have their source UML model element explicitly identified. Actors found in use case diagrams drive the overall page(s) to determine page's and section's ownership; Class attributes and associations drive section types and layout of components on that page; activity diagrams drive the sequences of interactions that comprise workflow based sections. Based on the experiments, H1 is considered to be at least partially supported, as the assignment of sections to pages draws on multiple diagrams when needed, and the proposed set of formal inference rules does generate an actor-based, traceable structure. Further, more broadly diverse testing should be completed to provide the statistical significance needed to strongly validate H1.

6.2.2 H2: Enterprise Application Suitability

The evaluation and prototype shows that, on prototype level, the system has potential for supporting multi-user and workflow-oriented enterprise application scenarios. Through Use Case diagrams, it is possible to derive which classifiers and operations are relevant for an actor and therefore, different actors may receive a different view on available actions and UI structures in the same system model [Wil24]. Additionally, activity diagrams allow inferring process-related information that helps shape workflows into suitable interfaces.
For example, workflow steps may map to specific UI section types such as forms for entry tasks or detail or list sections for review and selection tasks.
Workflow actions show navigation between workflow pages enabling the resulting prototype to reflect role-specific workflows. Nonetheless, the evaluation also demonstrated weaknesses in workflow handling, particularly in the case of more complex workflows. Hence, H2 is partially validated: The system has the potential to support enterprise workflows that involve multiple roles, yet there is a need for better guidance in the workflows.

6.2.3 H3: Human-in-the-Loop Effectiveness

The human-in-the-loop approach enables the designer to perform iterative exploration of the interface design by providing regenerated interface candidates based on selected candidates and natural language descriptions. Instead of generating UML-to-interface mappings anew from scratch, the candidate generation takes into consideration the pages, sections, layout settings, style and design tokens of a candidate to generate new candidate variants in the same Interface Metadata DSL. By doing this, it's possible to tune the appearance and layout preference, while keeping the original UML-based layout of the candidates being generated unchanged.
For example, a designer can suggest a more compressed layout, a new colour palette, or shifting regions in a candidate through textual commands, enabling quick generation of preview alternatives to choose from for comparisons of different designer intents [dH24].
Despite enabling iterative refinement with visual preview, the evaluation data reveals that human-in-the-loop refinement received one of the lowest scores on the detailed dimensions (and one of the

largest standard deviations). This indicates that this feature is usable, but its accuracy and level of control are not yet reliable; hence, we can only partially support H3.

6.3 Limitations

Here are some things to keep in mind as you consider the results. First, nine participants were recruited to evaluate our method, which is not large enough to generalize our results beyond the current context. We collected a convenience sample and our participants came with a variety of background knowledge (in low-code tools, UI design, software engineering) - the scores reflect that these are real world developers, but could be influenced by a participant's background, comfort level with the researcher, prior technology experience, and not solely the generated UI designs.
Second, only five of participants completed the full, detailed questionnaire, so our findings about role separation, workflow support, object centric navigation, candidate diversity, and refinement quality should be considered preliminary/exploratory.
Thirdly, one of the participants who played patient actor in Hospital case study scored very low on many projects, especially when testing the workflow. This may be due to the unclear content of the case study. Moreover, there are other limits to our evaluation. None of the participants completed multiple case studies from the beginning. This reflects that participants are more inclined to stick to the design phase, rather than making a complete prototype, showing that participants do not have the enthusiasm to test the system further.
Fourthly, a systematic audit of the rendering layer revealed that the Interface Metadata DSL's schema declares more style parameters than the Django/Jinja2 templates actually consume. Tracing every one of the 94 CSS custom properties declared in the shared stylesheet through the full chain — from its declaration, to the CSS class that reads it, to whether that class is ever emitted by an HTML template — showed that only 26 (28%) render unconditionally, 40 (43%) render only when a specific, sometimes rare, component or section type is present, and 28 (30%) are never applied to any rendered element regardless of input, most notably an entire typography scale (hero/display/lead/title sizing) and the semantic status-badge colors, which the badge macro does not consume. The same pattern recurs in the per-section `style` object: of 33 documented style fields, only 15 (45%) are read from more than one place in the main template, 13 (39%) are read from exactly one place and therefore depend on a specific component appearing, and 5 (15%) — including `seller_label` and `availability_label` — were not found to be read at all. Because the LLM is prompted against the full declared schema rather than this reduced, actually-functional subset, a candidate can be schema-valid and internally consistent while still failing to visually reflect part of the designer's request, independent of whether the LLM interpreted the request correctly. This is a plausible contributing factor to the comparatively low scores observed for candidate diversity and HITL refinement in the detailed evaluation (Section 5.3), and indicates that DSL-schema completeness and template-rendering completeness should be validated together rather than assumed to match.

6.4 Future Work

Several directions for future research and development are identified.

Integration with design tools and UI design artefacts. The current approach primarily draws from UML derived metadata, prompts and refinement feedback. In future work, direct import of designer artifacts from Figma, Penpot or other design system tooling would be beneficial. Design-to-code generation research has demonstrated that mockups and design artifacts can offer valuable structural and visual information for code generation [Bel18]. Within this thesis, imported design artifacts could be used beyond visual reference to supplement the structure provided by UML models by, for instance, enabling a correspondence between a designer's frame structure, component name, spacing, colour tokens and interaction hints, and our generated actors, domain objects and workflow classes, thus retaining designer intent whilst remaining tied to the software model.

Reverse engineering support for Django developers. The system, at this stage, enables forward generation of UI prototypes from UML models. The extended future work could include reverse engineering of current Django projects, too. A Django developer could input his existing project to the system.
The system will analyze Django's models, views, URL routes, forms, templates, and permissions and produce a class-like model, actors' responsibilities, page's composition.
According to MDRE, software understandability and evolvability may benefit from extracting high-level models from source code. The extended workflow will allow developers to visualize the reconstructed UML-like diagrams, update them, and thus generate a new design.

Connectors for external UML modelling tools. Another point of integration is importing models from existing UML modelling tools. Several organisations already make use of modelling tools to capture class diagrams, use case models, or activity models; however these models generally do not extend to the generation of executable UI prototypes. Hence in future developments, it might be possible to provide interfaces to popular model interchange formats like XMI (an OMG standard for exchanging metadata, including UML-related models, between UML tools [Obj26]), so that an existing UML model can be imported into the generator to generate prototypes tailored to specific users or workflows.

Stronger validation and formal constraints. Current workflow-centric approaches provide a mix of context structures and guardrails but lack a full formal constraint model. We could extend the Interface Metadata DSL to add first-class support for role-based access, object relationships, workflow ordering, data binding, and component compatibility constraints. The interface could be validated by static analysis tools to prevent rendering invalid components or referencing unavailable properties.

Broader evaluation. Finally, future work should evaluate the approach with a larger and more diverse participant group, including UI designers, Django developers, low-code users, and domain experts. Comparative studies against prompt-only generation, conventional low-code tools, or manual prototyping would provide stronger evidence about the benefits and limitations of the AI4MDE-oriented approach.

A LLM Prompt Templates

This appendix provides the full prompt templates used for candidate regeneration. Representative excerpts from candidate generation prompts are discussed in Section 4.4.1.

A.1 Base Styling Context (Regeneration)

The following template is injected into the regeneration prompt to anchor the LLM to the visual style of the selected candidate. Fields marked `<selected>` are populated at runtime from the chosen candidate's DSL styling object.

```
BASE CANDIDATE STYLING (starting point - inherit unless requirements
override): fontFamily=<selected>, textSize=<selected>,
accentColor=<selected>, backgroundColor=<selected>,
textColor=<selected>, radius=<selected>,
buttonStyle=<selected>, cardHover=<selected>
```

Listing 13: Base styling context injected into the regeneration prompt

A.2 Regeneration Diversity Constraints

The following prompt is injected alongside the base styling context
during regeneration. It ensures that the three regenerated candidates explore different design directions along axes not locked by the designer's feedback, while preserving the UML-derived section structure.

```
Generate exactly 3 meaningfully different refinements of the base
candidate. Each variant MUST differ on axes not locked by requirements:
  - typography (fontFamily, textSize)
  - density (compact vs normal vs spacious)
  - border radius (0 vs 8 vs 16 vs 24)
  - button style (solid vs outline vs ghost vs gradient)
  - card hover effect (lift vs glow vs border vs none)
  - page width (contained vs wide vs full)
  - nav placement (header vs sidebar)
Apply the DESIGNER REQUIREMENTS to all 3 variants.
Preserve base candidate's section roles and data bindings.
You may change layout/component/position for collection sections
but must keep object_form as form, object_detail as detail,
activity_* layouts unchanged.
```

Listing 14: Diversity constraints injected into the regeneration prompt

References

[ACM25] Matthew O. Ajimati, Noel Carroll, and Mary Maher. Adoption of low-code and no-code development: A systematic literature review and future research agenda. *Journal of Systems and Software*, 222:112300, 2025.

[ARNRSG06] Netta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[Bel18] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 3:1–3:6, 2018.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[CMW+24] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, Nan Duan, and Furu Wei. Low-code LLM: Graphical user interface over large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, pages 12–25. Association for Computational Linguistics, 2024.

[Col95] Dave Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, 1995.

[CSC25] Yunnong Chen, Chengwei Shi, and Liuqing Chen. SpecifyUI: Supporting iterative UI design intent expression through structured specifications and generative AI, 2025.

[CTBV23] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling*, 22(3):781–793, 2023.

[dH24] Pieter de Hoogd. An effective user experience for prototype generation in model-driven engineering. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2024.

[DMJ+24] Yi Dong, Ronghui Mu, Gaojie Jin, Yi Qi, Jinwei Hu, Xingyu Zhao, Jie Meng, Wenjie Ruan, and Xiaowei Huang. Position: Building guardrails for large language models requires systematic design. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 11375–11394. PMLR / OpenReview.net, 2024.

[Dri20] R. Driessen. UML class models as first-class citizen: Metadata at design-time and run-time. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2020.

[DWLH24] Peitong Duan, Jeremy Warner, Yang Li, and Bjoern Hartmann. Generating automatic feedback on UI mockups with large language models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 6:1–6:20. ACM, 2024.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[HZL+24] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):220:1–220:79, 2024.

[NMR24] Lukas Netz, Judith Michael, and Bernhard Rumpe. From natural language to web applications: Using large language models for model-driven software engineering. In *Proceedings of Modellierung 2024*, pages 179–195. Gesellschaft für Informatik e.V., 2024.

[Obj26] Object Management Group. Xml metadata interchange (xmi) specification, 2026. Accessed: 2026-06-24.

[PAA23] Daniel Pinho, Ademar Aguiar, and Vasco Amaral. What about the usability in low-code platforms? a systematic literature review. *Journal of Computer Languages*, 74:101185, 2023.

[RBW+25] Simon Rädler, Luca Berardinelli, Karolin Winter, Abbas Rahimi, and Stefanie Rinderle-Ma. Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering. *Software and Systems Modeling*, 24(2):445–469, 2025.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models of requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[vA22] Bram van Aggelen. Enabling data-driven wireframe prototyping using model-driven development. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2022.

[vD24] Sven Y. van Dam. A flexible, template-driven generation framework for model-driven engineering. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2024.

[Ver25] M. Verhoeff. Generating workflow-enabled prototypes from UML activity models for model-driven engineering. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2025.

[Wil24] L. Willemsens. Utilising use case models for multi-actor prototype generation. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2024.

[WSK+26] Jason Wu, Amanda Swearngin, Arun Krishnavajjala, Alan Leung, Jeffrey Nichols, and Titus Barik. Improving user interface generation models from designer feedback. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1162:1–1162:17. ACM, 2026.

[WSL+24] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P. Bigham, and Jeffrey Nichols. UICoder: Finetuning large language models to generate user interface code through automated feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7511–7525. Association for Computational Linguistics, 2024.

[YZY+23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the Eleventh International Conference on Learning Representations*. OpenReview.net, 2023.

[Zei23] S. Saqlain Zeidi. Synergizing UML class modeling and natural language to code conversion: A GPT-3.5-powered approach for seamless software design and implementation. Bachelor thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2023.