CHALMERS
UNIVERSITY OF TECHNOLOGY

Software to Generate Linux Executables for Sending Eiffel Events Based on JSON Schemas at Nexer

Schema-driven code generation using large language models

Bachelor's thesis in Computer Science and Engineering

ADRIAN BEHRAMI
LEON AGNEZON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Bachelor's thesis 2026

Software to Generate Linux Executables for Sending Eiffel Events Based on JSON Schemas

Schema-driven code generation using large language models

ADRIAN BEHRAMI
LEON AGNEZON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Software to Generate Linux Executables for Sending Eiffel Events Based on JSON Schemas
Schema-driven code generation using large language models
ADRIAN BEHRAMI
LEON AGNEZON

Supervisor: Sakib Sistek, Department of Computer Science and Engineering
Examiner: John Camilleri, Department of Computer Science and Engineering
Industry supervisors: Nikolai Dahlberg, Jewel Augustine, and Johan Hermansson, Nexer Engineering

Bachelor's thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Software to Generate Linux Executables for Sending Eiffel Events Based on JSON Schemas
Schema-driven code generation using large language models
ADRIAN BEHRAMI, LEON AGNEZON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The Eiffel Protocol is a set of JSON-based event types used for traceability in continuous integration and continuous delivery (CI/CD) pipelines. Each event type is formally defined by a JSON Schema, and multiple versions of each event exist. Creating sender programs that produce valid Eiffel events has traditionally been a manual and error-prone process, particularly given the large number of event–version combinations involved.

This thesis presents the design and implementation of a software system that automates the generation of Linux-executable command-line tools for creating, validating, and sending Eiffel events via RabbitMQ. The system uses a schema-driven approach where the official Eiffel JSON Schemas serve as the primary structural source of truth for code generation. Additional helper logic is used to handle compatibility differences, prompt construction, example generation, and runtime integration details that are not fully captured by the schemas alone. A Python-based orchestration pipeline constructs prompts from the schemas and uses the OpenAI GPT-4o language model to generate Rust source code for each event–version pair. The generated code is then compiled, executed, and validated against the corresponding schema in an iterative feedback loop.

The resulting system successfully generates and validates 223 sender programs covering all officially published Eiffel event types and schema versions. Each generated sender is packaged as an independent Rust binary crate with deterministic integration tests. A wrapper CLI tool provides both an interactive guided mode and a passthrough mode for automation. The complete system includes CI/CD pipelines for automated testing, schema synchronization, and release packaging via GitHub Actions.

The findings suggest that AI-assisted code generation combined with rigorous schema-based validation is a viable approach for producing correct, schema-conformant tools at scale, although the quality of the generated output is highly sensitive to prompt design.

Keywords: Eiffel Protocol, JSON Schema, code generation, large language models, Rust, CLI, continuous integration, RabbitMQ.

Acknowledgements

We would like to express our gratitude to our university supervisor, Sakib Sistek, and our examiner, John Camilleri, at the Department of Computer Science and Engineering at Chalmers University of Technology, for their guidance and feedback throughout this thesis.
We are grateful to our industry supervisors at Nexer Engineering, Nikolai Dahlberg, Jewel Augustine, and Johan Hermansson, for providing the project opportunity, sharing their expertise on the Eiffel Protocol and its ecosystem, and for their continued support during the development process.
We also thank the Eiffel Community for maintaining the open-source protocol specification and JSON Schemas that formed the foundation of this work.

Adrian Behrami and Leon Agnezon, Gothenburg, June 2026

List of Acronyms

Below is the list of acronyms used throughout this thesis, listed in alphabetical order:

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command-Line Interface
JSON	JavaScript Object Notation
LLM	Large Language Model
PR	Pull Request
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier

Contents

List of Figures

List of Tables

1
Introduction

This chapter introduces the context and motivation for the thesis, defines the purpose and research questions, and outlines the scope and structure of the report.

1.1 Background

Modern software development relies heavily on continuous integration and continuous delivery (CI/CD) pipelines to automate building, testing, and deploying software. As these pipelines grow in complexity, the need for end-to-end traceability across the various stages of the delivery process becomes increasingly important. Traceability enables development teams to understand what was built, how it was tested, and when it was deployed, which is essential for debugging, auditing, and compliance [1].

The Eiffel Protocol, originally developed by Ericsson, addresses this need by defining a set of event types that represent activities in a CI/CD pipeline [2]. Events such as "artifact created," "activity started," and "confidence level modified" can be published to a message broker to form a traceable chain of activities. Each event type is formally specified as a JSON structure, and the structural rules for each event and version are defined by JSON Schemas hosted in the Eiffel Community's open-source repository on GitHub [3].

The protocol currently defines 24 distinct event types, many of which have multiple schema versions. To send an Eiffel event programmatically, a developer must create a *sender*, a program that constructs a JSON object conforming to the appropriate schema and publishes it to a message broker such as RabbitMQ. Because each event type and version has a unique schema with different required fields, constraints, and structural rules, the manual creation of senders is a repetitive and error-prone process.

Prior to this thesis, the authors developed a Python-based tool in an earlier course project that could generate Python sender modules from Eiffel JSON Schemas using AI-assisted code generation [4]. While functional, this tool produced Python scripts rather than standalone executables, which limited its usefulness in Linux-based CI/CD environments where lightweight, dependency-free binaries are preferred. The industry partner for this thesis, Nexer Engineering in Gothenburg, identified the need for a Linux-native executable tool that could be distributed and used directly in CI/CD pipelines without requiring a Python runtime.

1.1.1 Practical Motivation and Problem Context

Modern CI/CD pipelines are typically composed of several tools responsible for different tasks, such as building, testing, packaging, and deployment. These tools often operate independently and may be developed by different vendors or teams. As a result, it can be difficult to reconstruct exactly what happened during a software delivery process, how different activities are related, and which outputs were produced at each stage.

The Eiffel Protocol addresses this problem by defining standardized events that describe what has happened in a pipeline. Instead of each tool reporting information in its own format, Eiffel provides a shared event model that can be used across systems. This makes it possible to connect activities such as a build, a test execution, an artifact publication, and a deployment into a traceable workflow.

A concrete example is a pipeline in which a source code change triggers a build, the build produces an artifact, the artifact is tested, and the tested artifact is then packaged and deployed. Without a standardized event structure, each step may be visible only inside its local tool, making cross-tool traceability difficult. With Eiffel, each of these stages can emit standardized events, allowing external systems to reconstruct the full history of the pipeline and understand how the stages are connected.

However, the existence of official Eiffel JSON Schemas does not automatically provide usable sender software. The schemas define how valid events should be structured, but they do not by themselves create Linux executables that can be run in practice. This means that organizations wishing to emit Eiffel events still need sender implementations that correctly match the schema requirements.

This becomes a practical challenge because Eiffel contains many event types and multiple schema versions. Implementing and maintaining sender programs manually for all of these combinations is repetitive, time-consuming, and error-prone. For this reason, automated generation of sender tools from the official schemas represents an important engineering problem with clear industrial relevance.

This thesis addresses that problem by developing a Linux-oriented pipeline that generates Rust-based Eiffel sender tools from official JSON Schemas, validates them, tests them, and prepares them for practical use.

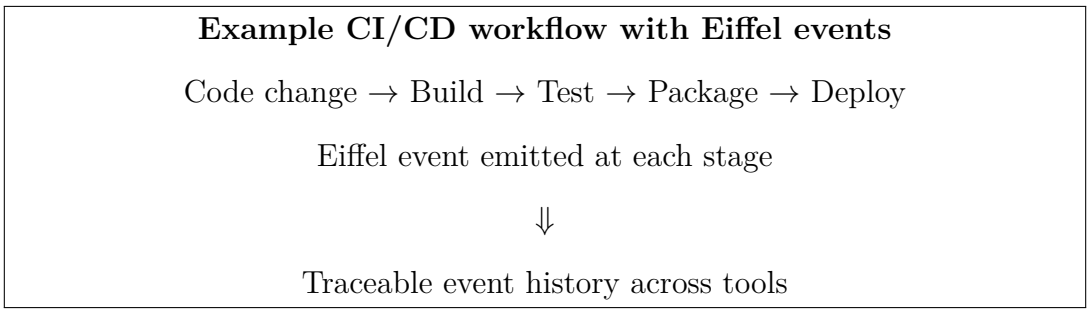

Figure 1.1: Illustrative example of how Eiffel events can be used to reconstruct a CI/CD workflow across multiple tools.

1.2 Purpose

The problem addressed in this thesis is therefore not the absence of event specifications, but the absence of practical sender tools that can reliably emit schema-compliant Eiffel events in a Linux environment. Although the official Eiffel JSON Schemas define the required event structures, they do not by themselves provide executable tools that organizations can use directly in CI/CD-related workflows.
The purpose of this thesis is to develop and evaluate a Linux-oriented pipeline for generating Eiffel event senders from official JSON Schemas. The work focuses on transforming schema definitions into usable Rust-based sender tools that can be validated, tested, and optionally used for message publishing through RabbitMQ.
More specifically, the thesis aims to investigate how schema-driven generation, AI-assisted code creation, validation, testing, and packaging can be combined into a practical engineering solution. A further purpose is to examine to what extent such a solution can support multiple Eiffel event types and versions while remaining maintainable and useful in an industrial setting.
The implementation presented in the following chapters therefore focuses on automating the generation, validation, testing, and usage of Linux-based Eiffel sender tools.

1.3 Research Questions

The thesis is guided by the following research questions:

RQ1: How can a schema-driven pipeline using AI-assisted code generation produce correct Eiffel event senders for all event types and schema versions?

RQ2: How can the correctness of AI-generated source code be systematically validated in the context of a large number of event–version combinations?

RQ3: What are the benefits and limitations of using large language models for structured code generation compared to deterministic, template-based approaches?

1.4 Contributions

This thesis makes both an engineering contribution and a scientific contribution.
The engineering contribution is the implemented Linux Eiffel sender system. This artefact consists of a schema-driven generation pipeline, AI-assisted Rust sender generation, automated validation and retry logic, generated sender crates, RabbitMQ publishing support, CI/CD automation, and a distributable Linux CLI wrapper. The artefact demonstrates that it is practically possible to generate and package sender tools for all evaluated Eiffel event types and schema versions.
The scientific contribution is the knowledge gained from designing, evaluating, and reflecting on this artefact. The thesis provides observations about how large language models behave in schema-driven code generation, especially when schemas vary across event types and versions. It shows that LLM-generated code cannot be trusted directly, but can be made usable through structured prompt construction, automated validation, deterministic helper logic, and retry mechanisms. The thesis

also identifies limitations related to reproducibility, model dependency, prompt sensitivity, and the boundary between schema-defined structure and implementation knowledge supplied by the surrounding generator.
Thus, the value of the work is not only the final tool, but also the evaluation of an engineering approach for combining formal schemas, LLM-based generation, and automated validation in a real CI/CD-oriented protocol setting.

1.5 Delimitations

The scope of this thesis is subject to the following delimitations:

- The system produces command-line tools only; no graphical user interface is developed.
- The generated executables target Linux only; other operating systems are not supported.
- Only officially published Eiffel event types and schema versions are supported. The Eiffel Schemas themselves are not modified.
- Performance optimization and benchmarking of the generated executables are outside the scope of this thesis.
- Event publishing is limited to RabbitMQ as the message broker.
- All test cases are human-written; no tests are generated by the AI model.

1.6 Ethical Considerations

The use of AI-assisted code generation in this project raises ethical considerations regarding transparency and correctness. Since the generated source code is produced by a large language model, it cannot be assumed to be correct by default. To mitigate the risk of deploying faulty code, the system employs automated validation against the official JSON Schemas, human-written system tests, and a code review process through pull requests on GitHub. Furthermore, the AI model is not permitted to generate or modify test cases, ensuring that the definition of correct behavior remains under human control.
The project does not involve the processing of personal data, and the generated tools operate on structured event metadata rather than sensitive information. The environmental impact of the project is limited to the computational cost of AI model inference during the generation phase, which is a one-time cost per event–version pair.

1.7 Report Structure

The remainder of this report is organized as follows:

- **Chapter 2** provides the theoretical background on the Eiffel Protocol, JSON Schema, code generation, large language models, and the other technologies used in this thesis.

- **Chapter 3** describes the development methodology, the analysis of the previous system, and the architectural decisions made during the project.
- **Chapter 4** presents the detailed design and implementation of the system, including the generation pipeline, validation pipeline, CLI wrapper, testing framework, and CI/CD automation.
- **Chapter 5** presents the results of the implementation, including schema coverage, test outcomes, and the final release artifact.
- **Chapter 6** discusses the findings in relation to the research questions, evaluates the strengths and limitations of the approach, and identifies directions for future work.
- **Chapter 7** summarizes the conclusions of the thesis.

2

Theory

This chapter provides the theoretical background necessary to understand the technologies, protocols, and concepts that underpin the thesis. It covers the Eiffel Protocol and its event model, JSON Schema as a specification language, approaches to code generation, the role of large language models, message brokering with RabbitMQ, the Rust programming language, CI/CD automation, and related work.

2.1 The Eiffel Protocol

The Eiffel Protocol is an event-based protocol designed to provide end-to-end traceability in CI/CD environments [1]. Originally developed by Ericsson, the protocol is now maintained as an open-source project by the Eiffel Community on GitHub [2]. The protocol defines a vocabulary of event types that represent activities, artifacts, and outcomes in a software delivery pipeline. By publishing these events to a shared message broker, organizations can reconstruct the full history of a software artifact from source commit to production deployment.

2.1.1 Event Structure

Each Eiffel event is a JSON object with a standardized top-level structure consisting of three main sections [3]:

- **Meta:** Contains identity and context information for the event, including a unique identifier (`id`), the event type (`type`), the schema version (`version`), a timestamp (`time`), and optional fields such as tags and source information. The `id` field is a version 4 UUID generated at event creation time.
- **Data:** Contains the payload specific to the event type. For example, an `EiffelArtifactCreatedEvent` includes data about the artifact's identity and file locations, while an `EiffelActivityCanceledEvent` includes a reason for cancellation. The structure and required fields of the `data` section are defined by the event's JSON Schema.
- **Links:** Represents typed relationships between the current event and other events. Each link consists of a type (e.g., `ACTIVITY_EXECUTION`, `CAUSE`) and a target, which is the UUID of the referenced event. Links enable the construction of directed graphs that capture the causal and contextual relationships between pipeline activities.

2.1.2 Versioning

Eiffel event types are versioned independently using semantic versioning. When the structure of an event type changes,for example, through the addition of new fields, changes to required properties, or modifications to constraint rules, a new schema version is published. The protocol currently defines 24 event types, many of which have undergone multiple revisions, resulting in over 200 distinct event–version combinations. Older schema versions are preserved, allowing consumers to process events produced by systems using different protocol versions [3].

2.2 JSON Schema

JSON Schema is a declarative specification language for describing the structure and validation constraints of JSON data [5]. It is widely used for API documentation, configuration validation, and data interchange formats. A JSON Schema document defines which properties a JSON object may or must contain, the data types of those properties, enumerated allowed values, and structural constraints such as nested objects and arrays.
In the context of the Eiffel Protocol, each event type and version is accompanied by a JSON Schema that serves as the authoritative definition of the event's structure [3]. These schemas specify required and optional fields, allowed value types, enumeration constraints, and structural nesting rules. For the purposes of this thesis, the schemas act as the primary structural source of truth from which event fields, required properties, value constraints, and validation rules are derived. However, the implementation also relies on supporting logic outside the schemas, including compatibility handling, example construction, prompt-building rules, and sender integration behavior.
Key JSON Schema features used in the Eiffel schemas include:

- `required`: specifies which properties must be present in the JSON object.
- `properties`: defines the expected keys and their corresponding type constraints.
- `enum`: restricts a property to a fixed set of allowed values.
- `$ref`: enables composition by referencing shared sub-schema definitions.
- `additionalProperties`: controls whether properties not listed in the schema are permitted.

2.3 Schema-Driven Code Generation

Schema-driven code generation is a software engineering approach in which source code is automatically derived from a formal specification [7]. Rather than manually implementing data structures, serialization logic, or validation routines, a code generator reads the specification and produces the corresponding implementation. This approach is well established in areas such as API client generation from OpenAPI specifications [6] and data serialization from Protocol Buffer definitions [8].
The principal advantage of schema-driven generation is that it reduces the risk of

specification drift: the generated code is guaranteed to reflect the current state of the schema. When the schema is updated, the code can be regenerated without manual intervention. This property is particularly valuable in the Eiffel context, where over 200 event–version schemas must be supported and new versions may be introduced at any time.
Schema-driven generation can be implemented using deterministic template-based approaches, where a fixed set of rules maps schema constructs to code patterns, or using generative approaches, where a model produces code from the schema and a set of instructions. This thesis employs a generative approach using a large language model, as discussed in Section 2.4.

2.4 AI-Assisted Code Generation

Large language models (LLMs) are neural network models trained on large corpora of text and code that can generate coherent natural language and source code from a textual prompt [10]. Models such as OpenAI's GPT-4o [11] have demonstrated the ability to produce syntactically correct and functionally plausible code in a variety of programming languages, including Rust, based on natural-language instructions combined with contextual data such as schemas or examples.

2.4.1 Prompt Engineering

The quality of LLM-generated code is highly sensitive to the design of the input prompt [12]. Prompt engineering refers to the practice of crafting detailed and precise instructions that guide the model toward producing the desired output. In the context of code generation, effective prompts typically include the full specification to be implemented, explicit rules about naming conventions and structural patterns, constraints on what the output must and must not contain, and examples of expected behavior.
Research has shown that vague or underspecified prompts often result in code that appears plausible but contains subtle errors, such as incorrect data types, missing edge-case handling, or structural violations [13]. Iterative prompt refinement, where the prompt is improved based on observed generation failures, is a common strategy for improving output quality.

2.4.2 The Generate–Validate–Regenerate Pattern

A widely adopted strategy for using LLMs in automated code production is the generate–validate–regenerate loop [15]. In this pattern, the model generates an initial version of the code, an automated validator checks the output for correctness, and if validation fails, the error information is fed back into the model as part of a revised prompt. This cycle continues for a bounded number of attempts. The pattern is analogous to test-driven development, where failing tests guide the developer toward a correct implementation, except that the "developer" is the language model.

This approach mitigates one of the principal limitations of LLM-based generation: the non-determinism and occasional incorrectness of model output. By bounding the number of retries and using structured error feedback, the system can recover from generation failures without human intervention in most cases.

2.5 Message Brokers and RabbitMQ

A message broker is a middleware component that facilitates asynchronous communication between software systems by receiving, routing, and delivering messages [16]. Message brokers decouple the producers and consumers of messages, enabling scalable and resilient distributed architectures.
RabbitMQ is an open-source message broker that implements the Advanced Message Queuing Protocol (AMQP) [17]. It supports multiple messaging patterns, including point-to-point queuing and publish/subscribe, and is widely used in enterprise and CI/CD environments. In the Eiffel ecosystem, RabbitMQ serves as the transport layer through which Eiffel events are published and consumed [1].
The key concepts relevant to this thesis are:

- **Exchange:** A routing entity that receives messages from producers and distributes them to queues based on routing rules.
- **Queue:** A buffer that stores messages until they are consumed.
- **Routing key:** A label attached to a message that the exchange uses to determine which queue(s) should receive it.
- **Connection and channel:** An AMQP connection is a TCP connection to the broker; channels are lightweight virtual connections multiplexed over a single TCP connection.

2.6 The Rust Programming Language

Rust is a systems programming language designed for performance, reliability, and memory safety without a garbage collector [18]. Rust programs are compiled ahead of time into native machine code, producing standalone binaries that can be executed on the target platform without requiring a runtime environment or external dependencies.
Several properties of Rust make it suitable for the generated Eiffel senders in this thesis:

- **Native compilation:** Rust produces self-contained Linux executables, which aligns with the thesis requirement for distributable, dependency-free CLI tools.
- **Strong type system:** Rust's type system catches many categories of errors at compile time, which serves as an additional validation layer for AI-generated code.
- **Ecosystem:** The Rust ecosystem provides mature libraries for command-line argument parsing (`clap` [19]), JSON manipulation (`serde_json` [20]), UUID generation (`uuid`), asynchronous programming (`tokio`), and AMQP communication (`lapin` [21]).

- **Cargo:** Rust's build system and package manager, Cargo, provides standardized project structure, dependency management, and a built-in test runner.

2.7 Continuous Integration and Continuous Delivery

Continuous integration (CI) is the practice of automatically building and testing software whenever changes are committed to a shared repository. Continuous delivery (CD) extends this by automatically packaging and preparing the software for release after all tests pass. Together, CI/CD pipelines reduce the time between writing code and delivering it to users while maintaining quality through automated gates.
GitHub Actions is a CI/CD platform integrated into GitHub that allows developers to define automated workflows triggered by repository events such as pushes, pull requests, or scheduled timers [22]. Workflows are defined in YAML files and consist of one or more jobs, each running in an isolated virtual environment. GitHub Actions supports matrix strategies for parallelizing jobs, service containers for dependencies such as databases or message brokers, and artifact management for build outputs.
In this thesis, GitHub Actions is used for three purposes: automated testing of generated senders on every code change, scheduled synchronization of upstream Eiffel schemas with automatic sender generation, and release packaging and publication.

2.8 Related Work

Several areas of prior work are relevant to this thesis.

2.8.1 Eiffel Community Tooling

The Eiffel Community maintains a set of open-source tools and libraries related to the protocol. These include the protocol specification and schemas [3], visualization tools, and integration examples. However, as of the time of writing, no existing tool automates the generation of standalone sender executables from the schemas. The existing tooling assumes that developers will implement senders manually or use language-specific libraries to construct events at runtime.

2.8.2 Schema-Based Code Generators

Code generation from schemas is a well-established practice. The OpenAPI Generator [7] produces API client libraries, server stubs, and documentation from OpenAPI specifications in over 40 programming languages. Google's Protocol Buffers [8] generate serialization code from `.proto` schema definitions. JSON Schema-specific generators such as `quicktype` [9] produce typed data structures from JSON Schemas in multiple languages. These tools use deterministic, template-based approaches. In contrast, this thesis uses a generative LLM-based approach, which offers greater

flexibility for producing complete, runnable programs rather than data structures alone, at the cost of non-deterministic output that requires validation.

2.8.3 LLM-Based Code Generation

Research on using large language models for code generation has grown rapidly. Studies have evaluated the correctness of LLM-generated code on benchmark suites [14] and explored techniques for improving generation quality through prompt engineering [12], chain-of-thought reasoning, and iterative refinement with test feedback [15]. The generate–validate–regenerate pattern used in this thesis is consistent with findings in this literature, which indicate that automated feedback loops significantly improve the success rate of LLM-based code generation.

3
Method

This chapter describes the methodology used to carry out the thesis work. It covers the development process, the analysis of the previous Python-based system, the key architectural decisions, the validation strategy, the evaluation criteria, and the tools and technologies employed.

3.1 Development Process

The project followed an agile development process characterized by iterative development cycles and close collaboration with the industry partner, Nexer Engineering. Regular meetings with the industry supervisors were held throughout the project to discuss progress, clarify requirements, and adjust priorities. The project ran from January 19 to June 26, 2026.
The work was organized into several phases, as illustrated in Figures 3.1, 3.2, and 3.3. The initial phase focused on studying the Eiffel Protocol, analyzing the existing Python-based tooling, and defining the architecture for the Linux-oriented system. The middle phase covered the main implementation work, including the generation pipeline, validation pipeline, and testing framework. The final phase addressed CI/CD automation, release packaging, and thesis report writing. Report writing was conducted in parallel with development throughout the project.
Figures 3.1, 3.2, and 3.3 together illustrate the project planning structure, including week ranges, task allocation, and overall timeline.

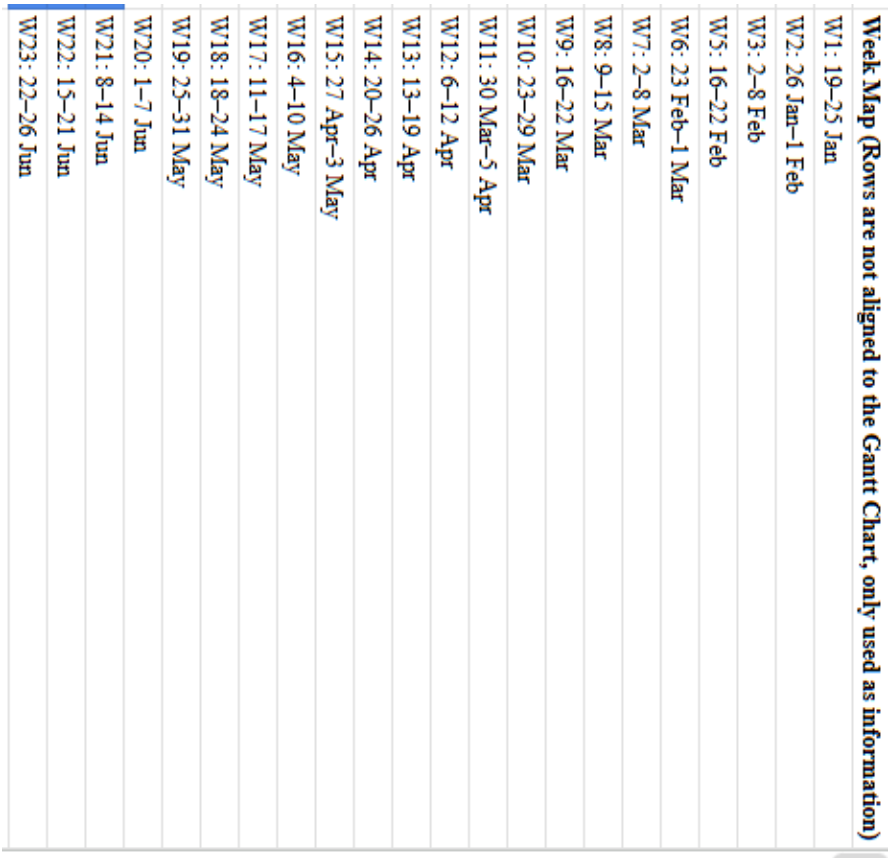

Figure 3.1: Week ranges used in the project timeline.

Students	Phase	Task
Leon Agnezon	1. Initiation & Planning	Supervisor communication, scope, planning report, repo + CI skeleton
Adrian Behrami	2. Study & Analysis	Study Eiffel Protocol + JSON Schemas, analyze schema/version structure
	2. Study & Analysis	Review previous Python tooling/workflows + identify integration points
Color Coding	3. Architecture & Design	Define architecture (schema ingest → codegen → validation → CLI → release)
Blue = Technical work	3. Architecture & Design	CLI UX + command structure + event/version selection design
Green = Documentation	3. Architecture & Design	Test strategy: system tests, fixtures, RabbitMQ test env, quality gates
	4. Implementation (Sprint 1)	Schema parsing + version handling module
	4. Implementation (Sprint 1)	Rust sender template generation + minimal payload builder
	4. Implementation (Sprint 2)	JSON Schema validation integration (generated sender validates payload)
	4. Implementation (Sprint 2)	RabbitMQ publish module + config handling
	4. Implementation (Sprint 2)	CLI: event/version selection wired to generated tool
	4. Implementation (Sprint 3)	System test framework + initial human-written test suite
	4. Implementation (Sprint 3)	AI "fix loop" for generation failures (code changes only, tests locked)
	4. Implementation (Sprint 3)	GitHub Actions CI: run tests on PR/push + prevent CI loops
	5. Testing & QA	Expand test coverage (invalid inputs, missing args, edge cases)
	5. Testing & QA	Regression testing across multiple events/versions
	6. Release & Documentation	Build Linux executable in CI + packaging
	6. Release & Documentation	GitHub Releases versioned artifacts + release notes process
	6. Release & Documentation	Documentation: README (user) + architecture/testing docs (dev)
	7. Finalization & defense prep	Final polish: verify, freeze scope, bugfix only
	7. Finalization & defense prep	Proofreading, formatting, supervisor feedback integration
	7. Finalization & defense prep	Presentation slides + rehearsal + defense prep
	8. Thesis writing (parallel)	Background/Theory draft
	8. Thesis writing (parallel)	Method + Implementation writing (as features land)
	8. Thesis writing (parallel)	Evaluation + ethical/societal + delimitations

Figure 3.2: Overview of students, project phases, and associated tasks.

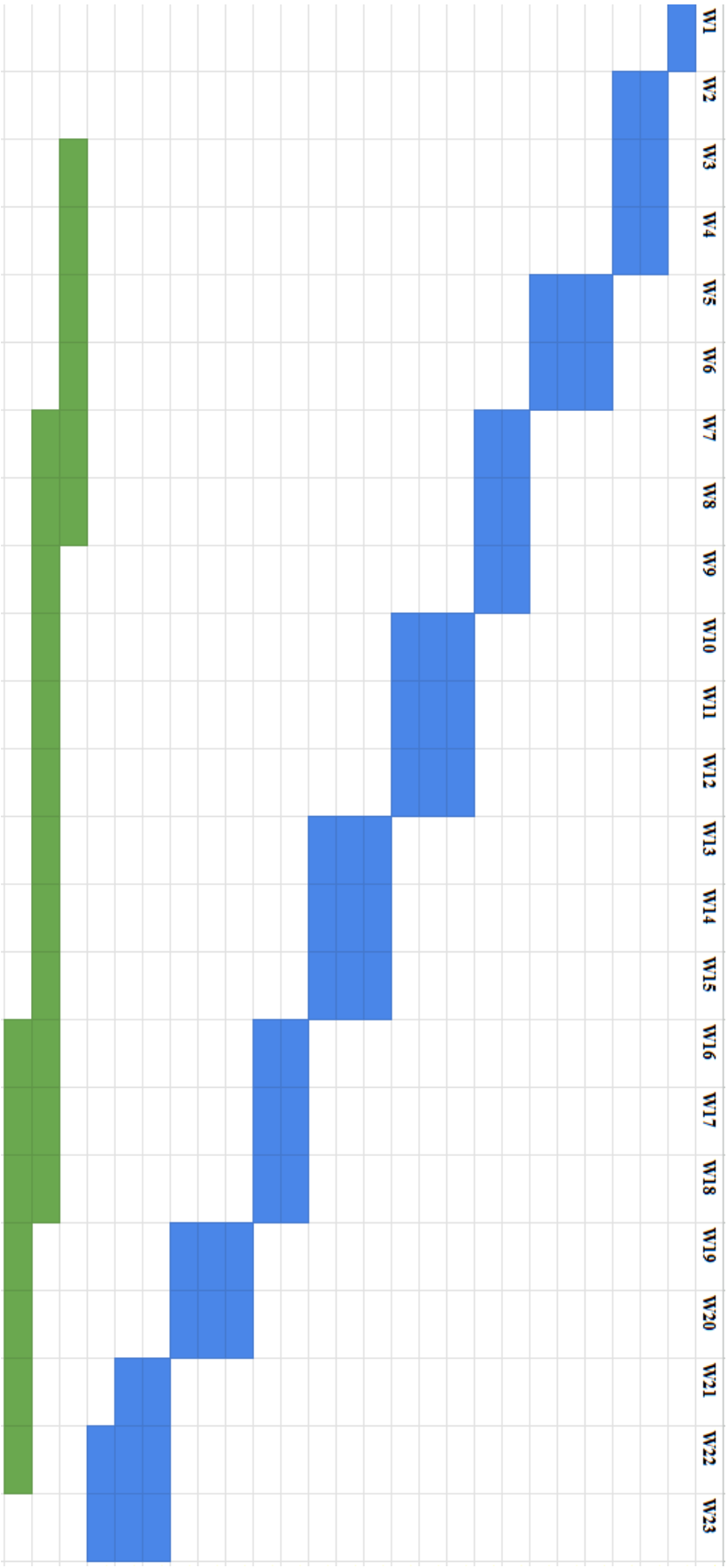

Figure 3.3: Gantt chart showing the project timeline and phase distribution.

3.2 Analysis of the Previous System

Since the thesis was a continuation of a previous course project by the same authors, the first major activity was to analyze the earlier Python-based implementation in sufficient depth to determine what could be reused and what needed to be redesigned.

3.2.1 Overview of the Previous System

The earlier system had been developed as a Python-based CLI module in which both the generation logic and the produced sender output were Python-based. The system already embodied a schema-driven philosophy: the official Eiffel JSON Schemas were used as the central input for generating senders, and AI-assisted generation via the OpenAI API was used to produce the sender code. The Python CLI supported listing available events, generating senders for specific event–version pairs, and running the generated senders to produce Eiffel event JSON.

3.2.2 Assessment for Reuse

A detailed review of the earlier codebase was carried out. The review examined how schemas were loaded, how prompt-based generation was organized, how sender files were written to disk, how validation was performed, and how the various utility modules interacted. This analysis was necessary because the thesis did not only involve implementing a new sender target; it also required deciding which parts of the existing system should remain responsible for orchestration and which parts should be redesigned.

The conclusion from the initial analysis was that the overall Python-based orchestration could be reused to a large extent, but that several core components required modification to support Rust-based Linux sender generation. In particular, the sender-writing logic, command-line support, validation flow, schema-based input generation, and testing strategy all needed to be adapted to the new target environment.

3.2.3 Identified Changes

The most significant difference between the two systems was the output format. The previous system produced Python files that could be executed directly by the Python interpreter. The Linux-oriented system needed to produce Rust binary crates that could be compiled into standalone executables. This change affected the file writer, the validator, the prompt builder, and the testing infrastructure.

Additionally, the earlier system's validation was tightly coupled to Python-side execution, which was not suitable for validating compiled Rust binaries. The testing approach also needed to be redesigned: instead of importing and calling Python functions, the tests had to treat each generated sender as a black-box command-line program.

The method used in the thesis therefore became one of *structured reuse combined with iterative redesign*: retaining the parts of the previous system that were target-

independent (schema loading, pipeline coordination, AI communication) while replacing or extending the parts that were specific to the output language and runtime.

3.3 Architectural Decisions

Several architectural decisions shaped the design of the system. These decisions were made early in the project based on the analysis of the previous system and the requirements from Nexer.

3.3.1 Dual-Language Architecture

The system uses two programming languages in complementary roles. Python serves as the orchestration and generation layer: it loads schemas, constructs prompts, communicates with the language model, writes generated code to disk, and runs validation. Rust is used for the generated output: each Eiffel event sender is a compiled Rust binary that can be executed directly on a Linux system.
This separation was chosen because Python's dynamic nature and rich library ecosystem make it well suited for tasks involving JSON processing, AI API interaction, and scripted automation, while Rust's ahead-of-time compilation produces the standalone, dependency-free executables required by the project specification.

3.3.2 Schema as Primary Structural Source

All event structure, field requirements, and validation rules are primarily derived from the official Eiffel JSON Schemas. The schemas provide the authoritative structural definition of each event type and version. However, the implementation also relies on supporting helper logic for compatibility handling, example generation, prompt construction, and sender integration behavior. This means that the system is schema-driven, but not schema-only.

3.3.3 One Crate per Event–Version

Each generated sender is an independent Rust binary crate with its own `Cargo.toml`, `src/main.rs`, and test files. This isolation ensures that different event–version pairs, which may have substantially different schema structures, do not interfere with each other during compilation or testing. It also enables each sender to be compiled, tested, and debugged independently.

3.4 Validation Strategy

Correctness of the generated senders is a central concern of the thesis. A generated sender is considered correct if it satisfies the following criteria:

1. It compiles successfully using the Rust compiler.
2. It executes without runtime errors when given valid input.
3. It produces syntactically valid JSON as output.

4. The produced JSON validates against the official Eiffel JSON Schema for the corresponding event type and version.

These four criteria are checked automatically by the validation pipeline during and after generation. The validation is performed at two levels: during the generation loop (where failed validation triggers a retry with error feedback) and during CI (where all senders are compiled, executed, and schema-validated as part of the test suite).

In addition to the automated validation, a multi-tier testing strategy was developed to verify the generated senders from different perspectives, as described in Section 4.6.

3.5 Evaluation Criteria

The system was evaluated against the following criteria:

- **Schema coverage:** The proportion of officially published Eiffel event–version combinations for which a valid sender is generated and passes all tests.
- **Test pass rate:** The percentage of generated senders that pass all three tiers of testing (dynamic validation, handwritten black-box tests, and failure-mode tests).
- **Cross-version compatibility:** The ability of the generation and validation pipeline to handle structural differences across different schema versions of the same event type.
- **End-to-end functionality:** Verification that the complete workflow—from schema to generated sender to RabbitMQ publication—operates correctly on a Linux system.

3.6 Tools and Technologies

Table 3.1 summarizes the tools and technologies used in the project.

Table 3.1: Tools and technologies used in the project.

Category	Technology
Programming languages	Python (orchestration), Rust (generated senders)
AI model	OpenAI GPT-4o (via API)
Protocols and standards	Eiffel Protocol, JSON Schema
Message broker	RabbitMQ (AMQP)
CI/CD	GitHub Actions
Testing	Human-written system tests, JSON Schema validation
Build system	Cargo (Rust), pip (Python)
Containerization	Docker, Docker Compose
Version control	Git, GitHub (pull request-based workflow)

4

Implementation

This chapter presents the detailed design and implementation of the system. It describes the overall system architecture, the generation pipeline, prompt construction, validation, the structure of generated senders, the wrapper CLI, the testing framework, CI/CD automation, release packaging, and schema synchronization.

4.1 System Architecture Overview

The system is organized into three main layers: the Python orchestration layer, the generated Rust sender layer, and the CI/CD automation layer. Figure 4.1 illustrates the high-level architecture. The Python orchestration layer is responsible for schema loading, prompt construction, AI communication, code generation, validation, and test execution. The Rust output layer contains the generated sender binaries and the wrapper CLI that provides the user-facing interface. External dependencies include the OpenAI GPT-4o API for code generation, RabbitMQ for event publishing, and GitHub Actions for CI/CD automation.

All source code is organized under the `Linux/` directory within the repository. The Python tooling resides in `Linux/tool_generator/`, the generated senders in `Linux/generated_senders/`, the wrapper CLI in `Linux/eiffel_sender_cli/`, and the local schema mirror in `Linux/schemas/`.

Figure 4.2 illustrates the full Linux solution, including schema input, AI-assisted generation, validation, CI/CD testing, release packaging, and runtime usage.

4.2 Generation Pipeline

The generation pipeline transforms an Eiffel JSON Schema into a compiled and validated Rust sender crate. The pipeline is implemented as a sequence of Python modules, each with a specific responsibility. Figure 4.3 illustrates the pipeline flow.

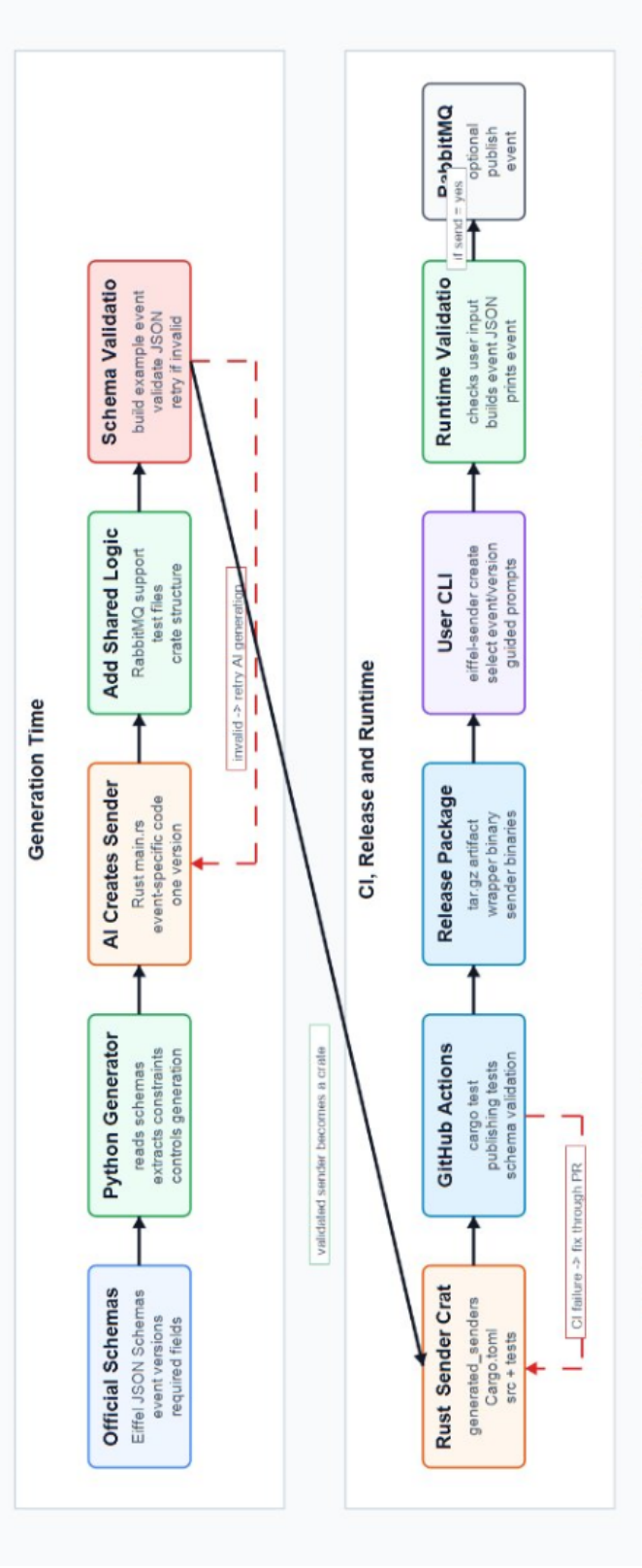

Figure 4.1: System architecture

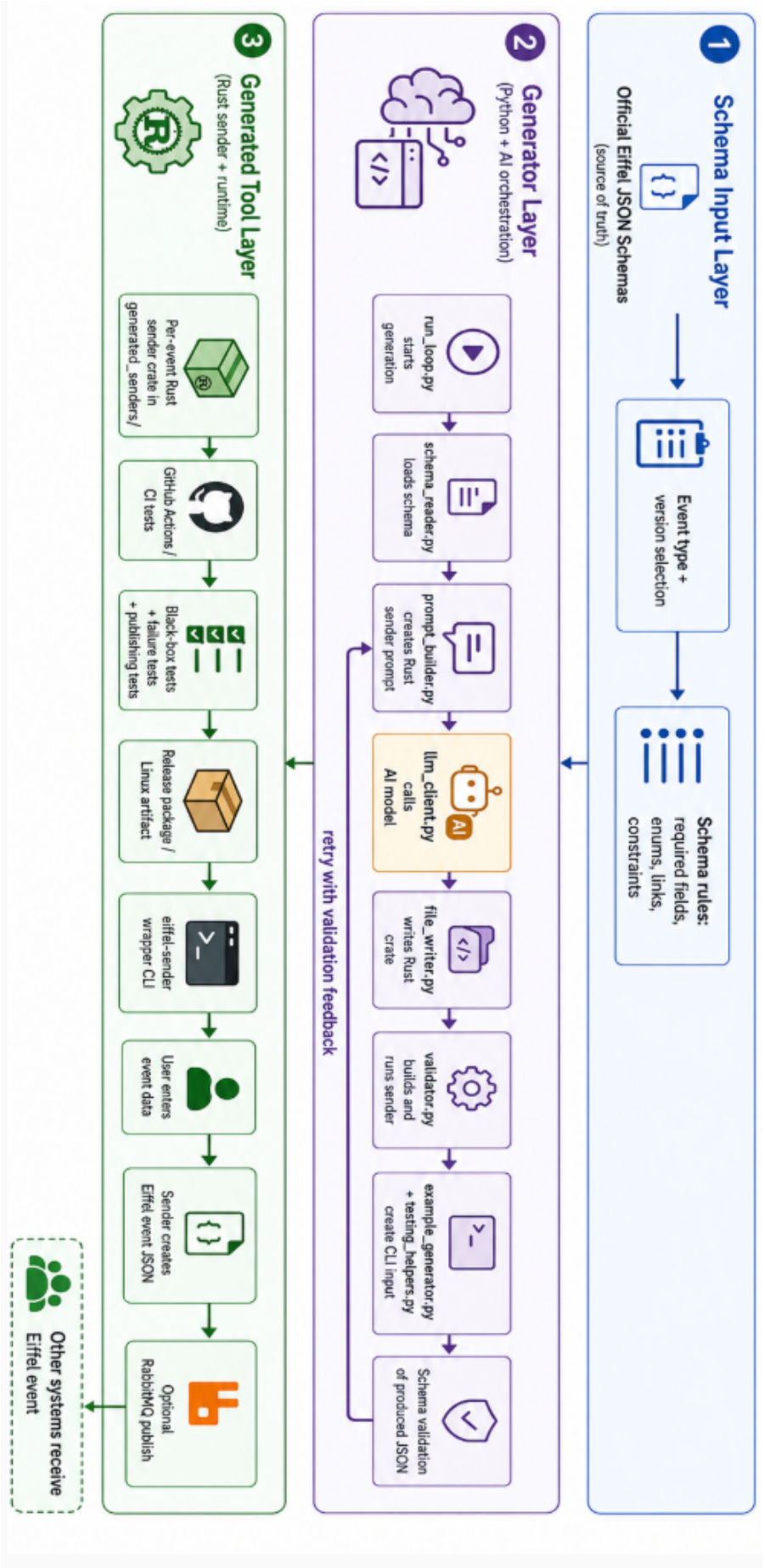

Figure 4.2: Overview of the Linux solution, including schema input, AI-assisted generation, validation, CI/CD testing, release packaging, and runtime event publishing.

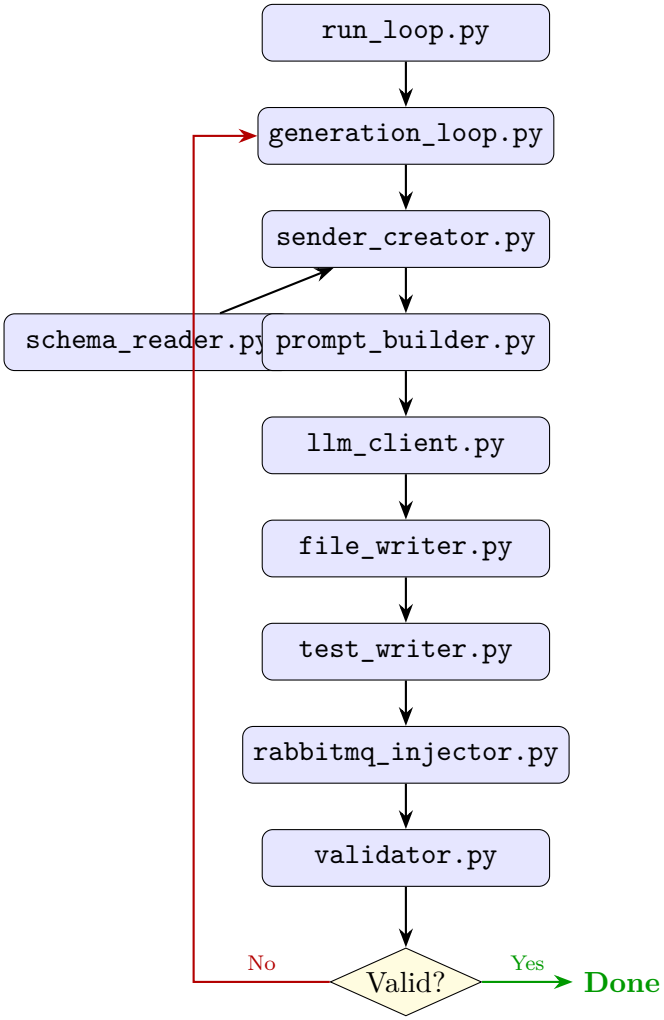

Figure 4.3: The generation pipeline, showing the sequence of modules and the validation feedback loop.

4.2.1 Entry Point and Retry Logic

The entry point of the generation process is `run_loop.py`, which receives the selected Eiffel event name, schema version, and a maximum number of allowed attempts from the command line. It delegates to `generation_loop.py`, which acts as the retry controller. For each attempt, `generation_loop.py` invokes the generation pipeline and then validates the result. If validation fails, the error information is stored and passed into the next generation attempt as feedback, creating the generate–validate–regenerate cycle described in Section 2.4. The default maximum number of attempts is five.

4.2.2 Schema Loading

The schema is loaded by `schema_reader.py`, which locates the appropriate JSON Schema file based on event name and version. The schemas are stored in a local mirror under `Linux/schemas/`, organized by event name and version number (e.g.,

schemas/EiffelActivityCanceledEvent/5.0.1.json). The schema is loaded into memory as a Python dictionary and serves as the primary specification for all subsequent stages.

4.2.3 Prompt Construction

The prompt sent to the language model is constructed by prompt_builder.py. This module translates the JSON Schema into a detailed natural-language specification for a Rust CLI program. The prompt includes:

- The full JSON Schema as an embedded reference.
- Rules for mapping schema field paths to clap command-line flags (e.g., data.reason becomes --data-reason).
- Structural requirements for the generated JSON output, including correct nesting and omission of optional fields when not provided.
- Instructions for handling Eiffel-specific constructs such as link types (formatted as TYPE=UUID), enumerated values, and arrays.
- Constraints on what the generated code must *not* do, such as producing null values, empty objects, or placeholder strings.
- Requirements for automatic generation of meta fields (UUID, timestamp, event type, and version).

The prompt evolved significantly during the project. Early versions contained only a general instruction to generate a Rust sender from the schema, which resulted in code that was often superficially plausible but contained subtle structural errors. Through iterative refinement driven by recurring validation failures, the prompt was strengthened with increasingly specific rules and prohibitions, substantially improving the success rate of first-attempt generation.

4.2.4 LLM Communication

The constructed prompt is sent to the OpenAI GPT-4o API through llm_client.py. This module handles the API call, receives the generated Rust source code, and strips any Markdown code fences that the model may include in its response. The module is intentionally kept simple and contains no Eiffel-specific logic, serving purely as the interface between the internal pipeline and the external language model.

4.2.5 File Writing and Post-Processing

The generated Rust source code is written to disk by file_writer.py, which creates a complete Rust binary crate structure under Linux/generated_senders/<EventName>/v<X>_<Y>_<Z>/. The crate includes a Cargo.toml file with the appropriate dependencies and a src/main.rs file containing the generated source code. After the source code is written, test_writer.py generates three deterministic integration test files for the crate (described in Section 4.6), and rabbitmq_injector.py post-processes the generated main.rs to add RabbitMQ publishing support. The injector adds the necessary Rust imports, RabbitMQ CLI configuration arguments, and an asynchronous publish_to_rabbitmq() function. This post-processing step

was introduced to keep the LLM prompt focused on event construction logic, while the messaging transport is handled by a controlled, deterministic transformation.

4.3 Validation Pipeline

The validation pipeline is implemented in `validator.py` and is invoked after each generation attempt. Its purpose is to determine whether the generated sender is functionally correct. The validation consists of four sequential stages:

1. **Compilation:** The generated Rust crate is compiled using `cargo build`. A compilation failure indicates syntax errors or type mismatches in the generated code.
2. **Execution:** The compiled sender is executed with a minimal set of command-line arguments derived from the schema. If execution fails (e.g., due to a runtime panic or incorrect argument parsing), the sender is considered invalid.
3. **JSON parsing:** The standard output of the executed sender is parsed as JSON. If the output is not valid JSON, the sender is rejected.
4. **Schema validation:** The parsed JSON is validated against the original Eiffel JSON Schema using the Python `jsonschema` library. This checks that all required fields are present, all values conform to their specified types and constraints, and no disallowed additional properties are included.

If any stage fails, the validator returns a structured error report that includes the stage at which failure occurred and the specific error message. This report is used by `generation_loop.py` as feedback for the next generation attempt, allowing the language model to address the specific failure in its revised output.

4.3.1 Schema-Based Example Generation

A practical challenge in the validation pipeline is that the generated sender is a CLI program that expects command-line arguments, not a function that can be called with a Python dictionary. To construct valid arguments for execution, the system uses `example_generator.py`, which inspects the schema and produces a minimal JSON object that satisfies the schema's required fields. This object is then translated into command-line arguments by `testing_helpers.py`, which flattens nested JSON paths into CLI flags, resolves link types, and handles special field formats.

The `testing_helpers.py` module evolved significantly during development as the system was extended to cover all event types and versions. Different event families required different fields (e.g., `data.reason`, `data.executor`, `data.outcome`), and older schema versions used different security field layouts. The module grew from a simple flattening utility into a compatibility-aware helper that supplements missing values, adds defaults for older schema structures, and resolves required link types.

4.4 Generated Sender Structure

Each generated sender is a standalone Rust binary crate. The directory structure is shown in Figure 4.4.

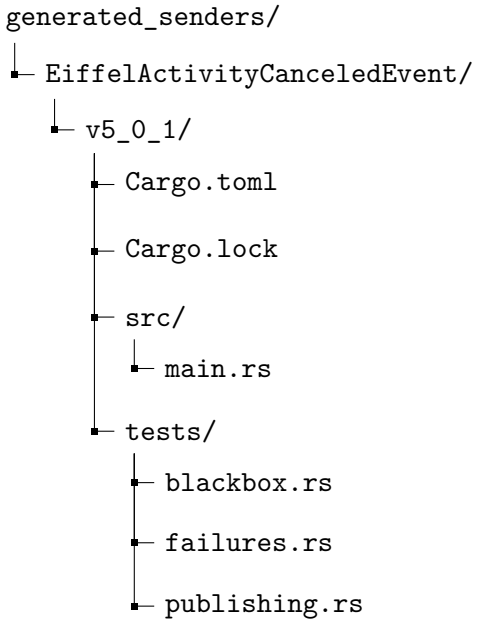

Figure 4.4: Directory structure of a generated sender crate.

The `Cargo.toml` file defines the crate name (derived from the event name and version), edition, and dependencies. The standard dependencies for all generated senders include `clap` for argument parsing, `serde_json` for JSON construction, `uuid` for generating the `meta.id` field, `chrono` for timestamps, `tokio` for asynchronous runtime, and `lapin` for AMQP communication with RabbitMQ.
The `src/main.rs` file contains the complete sender implementation. A typical generated sender follows this structure:

1. A `clap`-derived struct defining CLI arguments, with each required schema field mapped to a corresponding long flag.
2. Automatic generation of `meta` fields: a UUID for `meta.id`, the current UTC timestamp for `meta.time`, and hardcoded values for `meta.type` and `meta.version`.
3. Construction of the `data` section from CLI arguments, with optional fields included only when provided.
4. Parsing of link arguments from `TYPE=UUID` format into structured JSON objects.
5. Serialization of the complete event as JSON and output to standard output.
6. Optional RabbitMQ publishing when the `--send` flag is provided.

4.5 Wrapper CLI

The user-facing entry point is a Rust wrapper CLI called `eiffel-sender`, implemented in `Linux/eiffel_sender_cli/`. The wrapper provides two modes of operation:
Figure 4.5 illustrates this facade-based interaction model, where the `eiffel-sender` wrapper CLI provides a single user-facing entry point while hiding the event-specific sender selection, validation logic, and optional RabbitMQ publishing behind a unified interface.

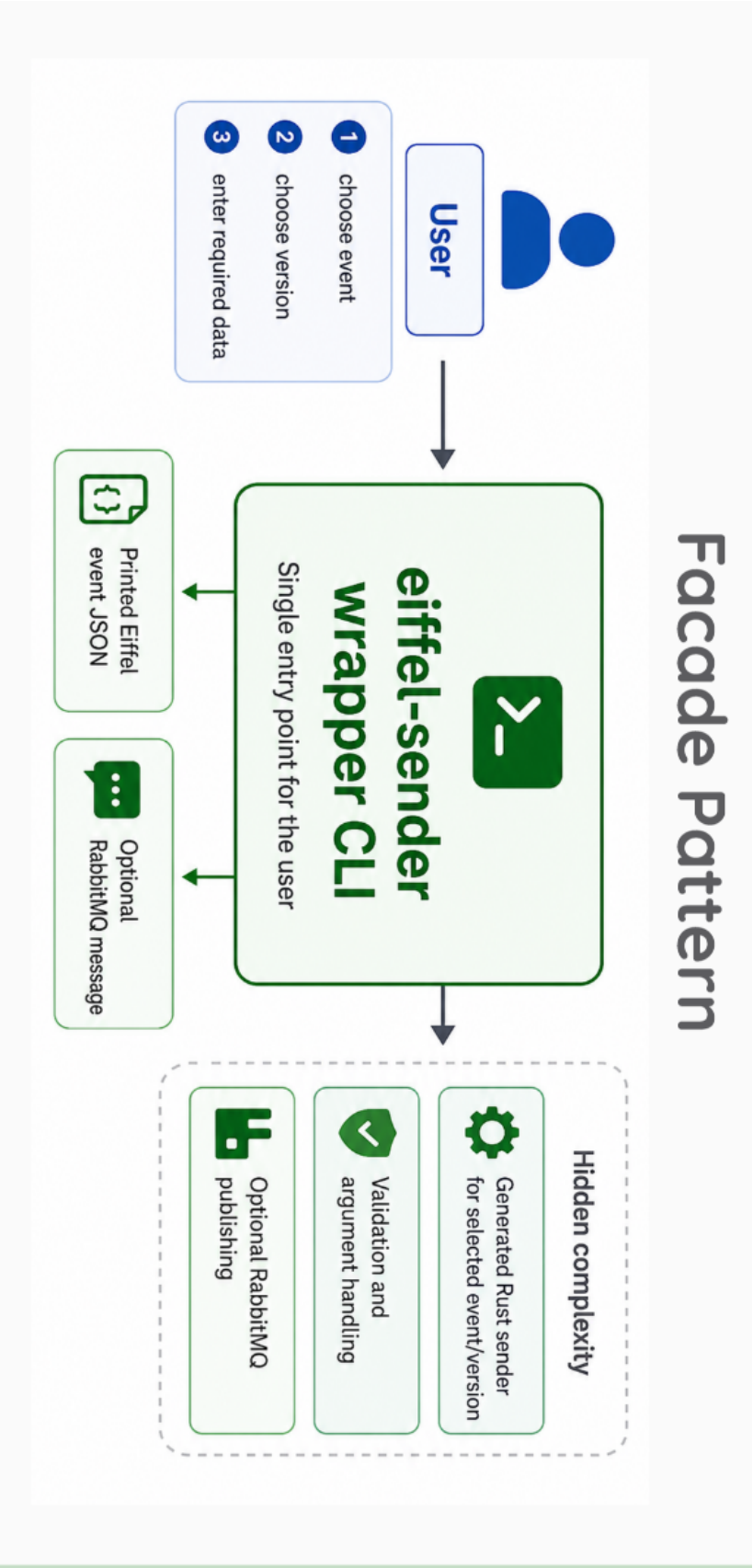

Figure 4.5: The `eiffel-sender` wrapper CLI acts as a facade that hides event-specific sender complexity behind a single user-facing interface.

4.5.1 Guided Mode

In guided mode, the user specifies an event type and version:

```
./eiffel-sender create --event <EventName> --version <Version>
```

The wrapper reads the corresponding schema from the packaged `schemas/` directory, extracts the required fields, and interactively prompts the user for each field value. The wrapper performs basic input validation, including checking enumerated values, validating UUID format for link targets, and supporting compact input formats for structured fields (e.g., `TYPE=URI` for location arrays). Once all fields are collected, the wrapper invokes the corresponding generated sender binary with the assembled arguments. If the user opts to publish the event to RabbitMQ, the wrapper also prompts for the broker URI and queue/routing-key configuration.

4.5.2 Passthrough Mode

Passthrough mode forwards all arguments directly to the selected generated sender without interactive prompting:

```
./eiffel-sender create --event <EventName> --version <Version> \
    -- --data-reason "Build failed" --links "CAUSE=<uuid>"
```

This mode is intended for automation and scripting scenarios where the caller provides all arguments programmatically.

4.6 Testing Framework

The testing strategy uses three complementary tiers to validate the generated senders from different perspectives.

4.6.1 Dynamic Validation Tests

The first tier is a dynamic automated test in `run_generated_rust_tests.py`. This test automatically scans the `generated_senders/` directory, discovers all Rust crates, compiles each one, executes it with automatically generated arguments, and validates the output against the corresponding schema. The test is parametrized and can be sharded for parallel execution in CI. Its primary advantage is scalability: it tests all senders without requiring a separate test file for each one.

4.6.2 Handwritten Black-Box Tests

The second tier consists of handwritten black-box tests in `test_handwritten_blackbox.py`. These tests use manually chosen CLI arguments for a representative subset of senders and verify the output structure independently of the automated helper logic. By not relying on `testing_helpers.py` for argument generation, these tests provide a more independent form of verification.

4.6.3 Failure-Mode Tests

The third tier verifies that generated senders correctly reject invalid input. Tests in `test_sender_failures.py` provide malformed or incomplete arguments and assert that the sender exits with a non-zero status code rather than producing incorrect output. This is an important aspect of CLI tool correctness: silent acceptance of invalid input would undermine trust in the generated events.

4.6.4 Rust-Side Integration Tests

In addition to the Python-based test tiers, each generated sender crate includes three Rust integration test files, generated deterministically by `test_writer.py`:

- `blackbox.rs`: Executes the sender binary with minimal valid arguments and verifies that the output JSON contains the expected `meta` fields (correct type, version, UUID format, and timestamp).
- `failures.rs`: Provides a malformed link argument and asserts that the sender exits with an error.
- `publishing.rs`: When a `RABBITMQ_URI` environment variable is set, publishes an event to RabbitMQ and verifies that the message is received in the expected queue with valid JSON content. This test is skipped when no RabbitMQ instance is available.

These tests are executed in a Docker-based Linux environment to match the intended runtime platform. They complement the Python-based tests by exercising the sender within Rust's own testing framework.

4.7 CI/CD Pipeline

Three GitHub Actions workflows automate the testing, schema synchronization, and release processes. Figure 4.6 summarizes the workflow triggers and responsibilities.

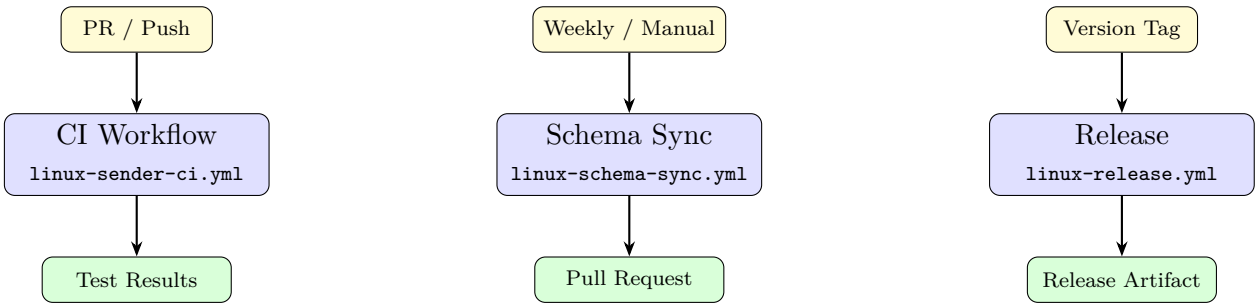

Figure 4.6: Overview of the three GitHub Actions CI/CD workflows.

4.7.1 CI Workflow

The CI workflow (`linux-sender-ci.yml`) runs on every pull request and push to the main development branches. It validates all generated sender crates by compiling them, running the Rust integration tests, and executing the Python-based schema validation. Because compiling all 223 senders in a single job would exceed GitHub

Actions runner resource limits, the workflow uses a matrix strategy to shard the validation across nine parallel jobs, each responsible for approximately 25 sender crates. Each shard starts a RabbitMQ service container to enable the publishing tests.

4.7.2 Schema Synchronization Workflow

The schema synchronization workflow (`linux-schema-sync.yml`) runs on a weekly schedule and can also be triggered manually. It fetches the latest schemas from the official Eiffel Community repository using the GitHub API, compares them with the local schema mirror, and identifies any new event types or versions. For each missing sender, the workflow invokes the generation pipeline with AI-assisted generation (requiring an `OPENAI_API_KEY` repository secret), runs the full test suite, and creates or updates a pull request on a dedicated branch. This design avoids direct commits to the main branch and ensures that all automatically generated changes undergo review before merging.

4.7.3 Release Workflow

The release workflow (`linux-release.yml`) is triggered by pushing a version tag (e.g., `v0.3.4`). It performs full validation of all senders, compiles them in release mode, and packages the release artifact using `package_linux_release.py`. The resulting archive (`eiffel-linux-senders.tar.gz`) contains the wrapper CLI binary, all generated sender binaries, and the schema files needed for guided mode. The artifact is uploaded to GitHub Releases for distribution.

4.8 Release Packaging

The release artifact is built by `package_linux_release.py`, which performs the following steps:

1. Compiles all generated sender crates and the wrapper CLI using Cargo.
2. Collects the compiled binaries into a structured directory layout.
3. Copies the schema files needed for the wrapper CLI's guided mode.
4. Creates a compressed archive (`.tar.gz`) of the release directory.

The resulting archive is self-contained: a user can extract it on any Linux system and immediately use the `eiffel-sender` tool without installing Rust, Python, or any other dependencies.

4.9 Schema Synchronization

To ensure that the system remains current with the upstream Eiffel Protocol, a schema synchronization mechanism was implemented. The module `fetch_schemas.py` uses the GitHub API to enumerate the schema files in the official Eiffel Community repository and download any new or updated schemas to the local mirror. The module `generate_missing.py` then compares the schema directory with the set of

generated senders and identifies any event–version combinations for which no sender exists.
For each missing sender, the generation pipeline is invoked with full validation. This process is bounded: if a sender cannot be generated within the maximum number of attempts, it is logged as a failure rather than blocking the synchronization. The workflow creates a pull request with all changes, ensuring that new senders are reviewed before being merged into the main branch.
This mechanism enables the system to adapt to protocol evolution without manual intervention, which is important given that the Eiffel Community periodically publishes new event types and schema versions.

5
Results

This chapter presents the results of the implementation. It reports on schema coverage, test outcomes across all tiers, cross-version compatibility, the final release artifact, and end-to-end RabbitMQ integration testing.

5.1 Schema Coverage

The system successfully generated and validated senders for all 223 event–version combinations present in the official Eiffel Community repository at the time of writing. This covers all 24 event types across their respective schema versions. Table 5.1 lists the event types and the number of schema versions for which senders were generated.

Table 5.1: Eiffel event types and the number of schema versions with generated senders.

Event Type	Versions
EiffelActivityCanceledEvent	9
EiffelActivityFinishedEvent	10
EiffelActivityStartedEvent	10
EiffelActivityTriggeredEvent	10
EiffelAnnouncementPublishedEvent	8
EiffelArtifactCreatedEvent	10
EiffelArtifactDeployedEvent	3
EiffelArtifactPublishedEvent	9
EiffelArtifactReusedEvent	8
EiffelCompositionDefinedEvent	9
EiffelConfidenceLevelModifiedEvent	11
EiffelEnvironmentDefinedEvent	9
EiffelFlowContextDefinedEvent	8
EiffelIssueDefinedEvent	7
EiffelIssueVerifiedEvent	11
EiffelSourceChangeCreatedEvent	9
EiffelSourceChangeSubmittedEvent	8
EiffelTestCaseCanceledEvent	8
EiffelTestCaseFinishedEvent	12
EiffelTestCaseStartedEvent	9
EiffelTestCaseTriggeredEvent	12

Event Type	Versions
EiffelTestExecutionRecipeCollectionCreatedEvent	12
EiffelTestSuiteFinishedEvent	11
EiffelTestSuiteStartedEvent	10

No event–version combinations were left without a generated sender, resulting in a schema coverage of 100%.

5.2 Test Results

All generated senders were tested across multiple tiers. Table 5.2 summarizes the test outcomes.

Table 5.2: Summary of test results across all testing tiers.

Test Tier	Passed	Failed	Total
Dynamic validation (compile + run + schema)	223	0	223
Rust blackbox tests	223	0	223
Rust failure-mode tests	223	0	223
Rust RabbitMQ publishing tests	223	0	223
Handwritten black-box tests (subset)	12	0	12
Handwritten failure-mode tests (subset)	12	0	12

All 223 generated senders pass the dynamic validation pipeline, which verifies compilation, execution, JSON parsing, and schema conformance. The Rust-side integration tests (blackbox, failure-mode, and RabbitMQ publishing) also pass for all senders. The handwritten Python-based tests, which cover a manually selected subset of 12 representative senders with independently chosen arguments, confirm the results.

5.3 Interpretation of Final Success Results

The reported result of 223/223 successful sender validations refers to the final evaluated state of the implemented Linux pipeline. More specifically, it means that all generated sender targets included in the final evaluation passed the implemented validation and testing flow at the time of measurement. This includes successful sender execution, JSON production, schema conformity, and, where applicable, the corresponding automated test stages used in the project.
It is important to clarify that this result does not imply that every sender was generated correctly on the first attempt during development, nor that the development process was free from failures, retries, prompt refinements, or compatibility adjustments. The final success result reflects the stabilized end state of the system after iterative development and refinement of the generation, validation, and helper logic.

The project method explicitly relied on iterative improvement. Generated senders were validated after creation, and failures were used as feedback for further refinement of prompts, helper logic, and compatibility handling. As a result, the final 223/223 result should be interpreted as evidence that the implemented pipeline was able to support all evaluated local schema targets in its final form, rather than as evidence that schema-to-sender generation succeeded immediately or without engineering intervention.

A limitation of the current evaluation is that detailed quantitative statistics regarding retry frequency, prompt iterations, generation cost, and average number of failed attempts were not systematically logged throughout development. This means that the thesis can report the final validated outcome clearly, but cannot provide a complete quantitative account of how much iteration was required to reach that state. This should be taken into account when interpreting the reported success rates.

5.4 Cross-Version Compatibility

Achieving full coverage required significant compatibility work to handle structural differences between schema versions. The main categories of differences encountered were:

- **Security field layout:** Newer schemas use a `meta.security` object with an `authorIdentity` field and optional integrity protection. Older schemas use a simpler `meta.security.sdm` structure. The testing helpers were extended to detect the schema version and supply the appropriate security fields.
- **Required data fields:** Different event families require different data fields. For example, activity events require a `data.reason` field, while artifact events require fields such as `data.identity` and `data.fileInformation`. The example generator and testing helpers were iteratively extended to cover these event-specific requirements.
- **Link type requirements:** Some event types require at least one link of a specific type (e.g., `ACTIVITY_EXECUTION` or `CAUSE`). The helpers were updated to detect required link types from the schema and include them in the generated test arguments.
- **Nested data structures:** Certain event types, particularly in newer versions, include deeply nested objects within the `data` section (e.g., locations with type and URI, outcomes with verdict and description). These are represented as compact `KEY=VALUE` pairs in the CLI, which required dedicated parsing logic in both the generated senders and the testing helpers.

The compatibility work was iterative: each time a new event type or version was added to the test suite, failures exposed missing handling in the prompt construction, example generation, or testing helpers. The relevant modules were then updated and all senders retested.

5.5 Release Artifact

The final release candidate is version 0.3.4, packaged as `eiffel-linux-senders.tar.gz`. The archive contains:

- The `eiffel-sender` wrapper CLI binary.
- All 223 generated sender binaries, organized by event type and version.
- The JSON Schema files required for the wrapper CLI's guided mode.
- A release README with usage instructions.

The archive is self-contained and can be extracted and used on any Linux system without additional dependencies.

5.6 End-to-End RabbitMQ Testing

End-to-end testing was performed manually using the release artifact in a Docker environment on Windows. The test procedure was as follows:

1. The release archive was extracted and mounted into an Ubuntu Docker container.
2. A RabbitMQ instance was started on the host system using Docker Compose.
3. The wrapper CLI was used in guided mode to create an `EiffelConfidenceLevelModifiedEvent` version 4.1.0, with required fields entered interactively.
4. The event was published to RabbitMQ using the host URI (`host.docker.internal`).
5. The RabbitMQ management UI confirmed that the event payload was received in the configured queue.
6. A second event (`EiffelArtifactPublishedEvent` version 4.0.1) was tested similarly, which led to improvements in the guided prompts for structured fields.

Both events were successfully published and received by RabbitMQ with valid JSON payloads that conformed to their respective schemas.

5.7 CLI Interaction Example

The following listing shows an example interaction with the wrapper CLI in guided mode, creating an `EiffelActivityCanceledEvent`:

```
$ ./eiffel-sender create \
    --event EiffelActivityCanceledEvent \
    --version 5.0.1

Required field: data.reason
> Build was superseded by newer commit

Optional field: meta.tags (comma-separated, or skip)
>

Optional field: links (TYPE=UUID, or skip)
> ACTIVITY_EXECUTION=550e8400-e29b-41d4-a716-446655440000

Publish to RabbitMQ? (y/n): n

{
  "meta": {
    "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
    "type": "EiffelActivityCanceledEvent",
    "version": "5.0.1",
    "time": 1716100000000
  },
  "data": {
    "reason": "Build was superseded by newer commit"
  },
  "links": [
    {
      "type": "ACTIVITY_EXECUTION",
      "target": "550e8400-e29b-41d4-a716-446655440000"
    }
  ]
}
```

6

Discussion

This chapter discusses the results in relation to the research questions, evaluates the strengths and limitations of the approach, identifies threats to validity, and suggests directions for future work.

6.1 Answering the Research Questions

RQ1: How can a schema-driven pipeline using AI-assisted code generation produce correct Eiffel event senders for all event types and schema versions?

The results demonstrate that a pipeline combining JSON Schema analysis, structured prompt construction, LLM-based code generation, and iterative validation can produce correct senders for all 223 event–version combinations. The key enablers of this result are threefold. First, using the schema as the primary structural source of truth ensures that generated code is derived from the authoritative event specification, reducing the risk of specification drift for event structure and validation constraints. At the same time, the schemas alone were not sufficient to drive the full implementation: helper modules were required to interpret schema-version differences, construct usable prompts, generate valid example inputs, and handle integration behavior outside the event schema itself. Second, the prompt construction module translates schema-level information into detailed, rule-based instructions for the language model, which proved essential for achieving reliable output. Third, the generate–validate–regenerate loop provides an automated recovery mechanism for generation failures, reducing the need for manual intervention.

The schema-driven approach also provides scalability: when new schemas are published, the pipeline can often generate corresponding senders without changes to the core pipeline code, provided that the new schemas follow patterns already handled by the existing helper logic. The schema synchronization workflow automates this process end to end, from fetching new schemas to generating senders and creating a pull request.

RQ2: How can the correctness of AI-generated source code be systematically validated in the context of a large number of event–version combinations?

Correctness is ensured through a four-stage validation pipeline (compilation, execution, JSON parsing, and schema validation) combined with a multi-tier testing strategy. The dynamic validation tests provide broad coverage across all 223 senders without requiring individual test files. The handwritten black-box tests provide independent verification for a representative subset. The failure-mode tests verify that invalid input is correctly rejected. The Rust-side integration tests complement the Python-based tests by exercising the senders within their native runtime environment.

This layered approach addresses different categories of potential errors: compilation failures indicate syntactic or type-level problems in the generated Rust code; execution failures reveal runtime issues such as panics or incorrect argument parsing; JSON parsing failures catch cases where the output is not well-formed; and schema validation failures detect structural or semantic violations of the Eiffel specification. The use of structured error feedback in the retry loop means that when validation fails, the specific nature of the failure guides the next generation attempt, which is more effective than simple retry without feedback.

RQ3: What are the benefits and limitations of using large language models for structured code generation compared to deterministic, template-based approaches?

The LLM-based approach offers several benefits compared to template-based generation. It can produce complete, runnable programs rather than data structures alone, handling the full range of code patterns needed for a CLI application (argument parsing, JSON construction, conditional inclusion of optional fields, link parsing, etc.). It adapts to the specific structure of each schema without requiring explicit template rules for every possible schema construct. It also proved effective at producing idiomatic Rust code that integrates well with the Rust ecosystem libraries.

However, the approach has notable limitations. The output of the language model is non-deterministic, which means that the same prompt may produce different code on different runs, and some runs may produce incorrect code. This non-determinism necessitates the validation pipeline and retry loop, adding complexity that a deterministic template-based approach would not require. Additionally, the quality of the generated code is highly sensitive to the design of the prompt: vague or incomplete prompts consistently produced subtly incorrect output, and significant effort was required to refine the prompt to the level where first-attempt generation success became reliable.

A template-based approach would offer full determinism and reproducibility, but would require implementing explicit templates for every structural pattern present in the Eiffel schemas, including complex nested objects, varied link types, and version-specific field layouts. Given the diversity of schema structures across 223

event–version combinations, the template development effort would be substantial. The LLM-based approach trades reproducibility for flexibility, with the validation pipeline serving as the quality gate.

6.1.1 Limitations of the Comparison to Template-Based Approaches.

An important limitation of this thesis is that no deterministic template-based baseline generator was implemented and evaluated in parallel with the LLM-based approach. As a result, the thesis does not provide an empirical head-to-head comparison of correctness, development effort, maintainability, or long-term robustness between these two approaches.
The observations made regarding the benefits and limitations of LLM-based generation are therefore based on the implemented Linux system, the engineering experience gained during development, and the behaviour of the final validated pipeline. These observations are still valuable for understanding the practical role of LLMs in schema-driven code generation, but they should not be interpreted as experimental proof that the LLM-based approach is superior to a deterministic template-based alternative.
It should also be noted that the thesis was carried out within the framework of an industrially motivated project specification in which AI-assisted generation of Linux sender tools was part of the defined assignment. The project description explicitly required the students to create software that generates Linux executable CLI tools using AI, together with surrounding validation, testing, release, and integration support. As a result, the practical focus of the thesis was to design, implement, and evaluate one complete AI-assisted generation pipeline rather than to develop multiple competing generator architectures in parallel.
This context helps explain why no deterministic template-based baseline was implemented as a separate system during the project. The thesis can therefore discuss the observed strengths and limitations of the implemented LLM-based approach, but it does not claim to provide a controlled experimental comparison against a fully developed template-based alternative.
The thesis can therefore support the conclusion that LLM-based schema-driven generation was viable in the implemented setting, but it cannot establish comparative superiority over non-LLM methods without further baseline development and evaluation.

6.2 AI-Generated Code Quality

Several observations about the quality of LLM-generated code emerged during development.
In the early stages of the project, when the prompt was relatively simple, the generated code frequently contained subtle errors: incorrect `clap` flag naming, malformed JSON nesting, inclusion of null values or empty objects that violated the schema, and incorrect parsing of link target strings. These errors were not immediately

apparent from reading the code, as the overall structure appeared plausible. This finding is consistent with the literature on LLM code generation, which reports that models often produce code that is syntactically correct but semantically flawed [13].

As the prompt was iteratively refined with more explicit rules and prohibitions, the frequency of these errors decreased substantially. This suggests that the limiting factor for generation quality in this context is prompt specificity rather than model capability. The model demonstrated the ability to produce fully correct senders for complex schemas when given sufficiently detailed instructions.

The decision to keep RabbitMQ integration out of the LLM-generated code and instead inject it through a deterministic post-processing step proved beneficial. The transport logic is structurally uniform across all senders, making it well suited for deterministic transformation, while the event-specific construction logic is more variable and benefits from the flexibility of LLM generation.

6.2.1 Limits of Schema-Based Correctness.

An important consideration in the evaluation is that schema validation, even when combined with executable and automated testing, does not by itself guarantee every aspect of software quality. In the implemented pipeline, generated senders are not only checked against the corresponding Eiffel schema, but also built, executed, and verified through several test layers. This provides strong evidence that the generated senders are structurally correct and practically usable within the evaluated workflow.

At the same time, these results should primarily be interpreted as evidence of structural correctness, executable behaviour, and conformance to the implemented validation and test criteria. They do not by themselves prove that every generated sender is optimal in all possible quality dimensions, such as long-term maintainability, or behaviour under all conceivable runtime conditions.

The results therefore demonstrate that the implemented pipeline can generate schema-compliant and runnable senders with strong practical verification, while still leaving some broader software-quality aspects outside the scope of the present evaluation.

6.3 Scalability

The system's scalability derives primarily from the schema-driven design. Adding support for a new event type or schema version is primarily driven by the schema, since the schema provides the structural event definition used for generation, validation, and testing. However, this assumes that the existing helper logic can interpret the schema pattern; unusual new schema constructs may still require updates to compatibility handling, prompt construction, or example generation. The schema synchronization workflow automates this end to end.

Practical scalability challenges emerged in CI/CD: compiling 223 Rust crates in a single GitHub Actions job exceeded available runner resources. The sharding strategy, which divides the crates across nine parallel jobs, addressed this problem, but would need to be extended as the number of senders grows.

6.4 Limitations

Several limitations of the current system should be noted.

Complex nested fields. Some Eiffel event types, particularly in newer schema versions, include deeply nested data structures with multiple levels of objects and arrays. The current CLI interface represents these as compact `KEY=VALUE` encodings, which can be unintuitive for users unfamiliar with the schema structure. A full interactive form interface or a JSON file input mode would provide a better user experience for these cases.

Single LLM provider. The system currently depends on OpenAI's GPT-4o model via the OpenAI API. A change in API availability, pricing, or model behavior could affect the generation process. The system does not currently support alternative LLM providers, although the `llm_client.py` module could be extended to support additional backends.

Linux only. The generated executables target Linux only, as required by the project specification. Cross-compilation for other platforms is technically possible with Rust but was not implemented or tested.

No runtime schema validation. The released sender binaries do not perform full JSON Schema validation at runtime. Validation is performed during the generation and testing phases, and the guided CLI wrapper performs basic input validation, but the sender binaries themselves trust that their input conforms to the expected format. This is a deliberate design choice to keep the binaries lightweight, but it means that passthrough mode does not guard against all forms of invalid input.

6.4.1 Reproducibility of the Generation Process

A central reproducibility limitation of the system is its dependency on the OpenAI GPT-4o model. The generation stage is not fully deterministic: even when the same schema, prompt, and repository state are used, the language model may return different Rust implementations across separate runs. This means that the exact generated source code cannot be guaranteed to be reproducible in the same way as code produced by a deterministic template generator.

This limitation is also affected by model evolution over time. Since GPT-4o is accessed through an external API, its behavior may change as the provider updates the model, serving infrastructure, safety filters, or default generation behavior. As a result, a generation run performed at a later date may not produce byte-for-byte identical sender code, even if the project code and Eiffel schemas remain unchanged. The system therefore depends not only on the repository state, but also on the availability and behavior of the external model at the time of generation.

However, not all parts of the system are equally non-reproducible. The input side is reproducible: the Eiffel JSON Schemas, prompt construction logic, validation code, test code, and generated release structure are stored in the repository. The acceptance criteria are also reproducible: a sender must compile, run, produce JSON, and validate against the corresponding schema before it is accepted. The final committed sender artifacts are reproducible as software artifacts in the sense that they can be rebuilt and retested from the repository.

What is not fully reproducible is the stochastic path by which a new sender is generated. The number of attempts, the exact intermediate failures, and the exact source code returned by the model may vary between runs. For this reason, the project treats LLM output as an untrusted candidate rather than as a reproducible build product. Reproducibility is instead achieved at the validation and artifact level: the system does not require the model to produce the same code every time; it requires any accepted code to satisfy the same schema-driven correctness checks. This distinction is important for interpreting the results. The thesis demonstrates that the pipeline can repeatedly produce valid Eiffel senders under the evaluated conditions, but it does not claim that the LLM generation step itself is deterministic. A more reproducible future design could reduce this dependency by recording model versions and generation parameters, caching accepted outputs, supporting multiple LLM providers, or replacing parts of the generated code with deterministic templates.

6.5 Threats to Validity

6.5.1 Internal Validity

The primary internal validity concern is the non-determinism of LLM-generated code. Because the model may produce different output for the same prompt across runs, the current set of generated senders represents one instance of a stochastic process. However, this concern is mitigated by the validation pipeline: all senders included in the system have passed all validation stages and tests, regardless of how many generation attempts were required.
The test coverage, while comprehensive across the three tiers, does not cover every possible combination of optional field values and input formats. The dynamic tests use minimal schema-conformant input, and the handwritten tests cover only a representative subset of senders.

6.5.2 External Validity

The approach was developed and evaluated specifically for the Eiffel Protocol and its JSON Schemas. The generalizability of the findings to other schema-based systems (e.g., OpenAPI specifications, Protocol Buffer definitions) has not been empirically evaluated, although the underlying principles, schema-driven prompt construction, iterative validation, and multi-tier testing, are not inherently Eiffel-specific.
The reliance on GPT-4o means that the results may not directly transfer to other language models, which may have different strengths and weaknesses in code generation. The effectiveness of the prompt is likely model-dependent.

6.6 Future Work

Several directions for future work have been identified:

- **Interactive form interface:** Developing a more sophisticated input mechanism for complex nested fields, such as a JSON file input mode or a terminal-based form interface, would improve usability for event types with deep data structures.
- **Alternative LLM providers:** Supporting multiple LLM backends (e.g., Anthropic Claude, open-source models) would reduce vendor dependency and enable comparison of generation quality across models.
- **Cross-platform support:** Extending the release pipeline to cross-compile for macOS and Windows would broaden the tool's applicability beyond Linux-based CI/CD environments.
- **Template-based hybrid:** Combining deterministic template generation for the structural boilerplate (argument parsing, JSON serialization, RabbitMQ publishing) with LLM generation for the schema-specific event construction logic could reduce the dependency on the language model while retaining the flexibility of the current approach.
- **Runtime schema validation:** Adding optional runtime validation against the JSON Schema in the sender binaries would provide an additional safety net for passthrough-mode users, at the cost of increased binary size and execution time.
- **Eiffel Community integration:** Publishing the tool through official Eiffel Community channels would enable broader adoption and community feedback.

7
Conclusion

This thesis presented the design and implementation of a software system that automates the generation of Linux-executable command-line tools for creating, validating, and sending Eiffel events. The system uses a schema-driven approach in which the official Eiffel JSON Schemas serve as the primary structural source of truth, combined with AI-assisted code generation using the OpenAI GPT-4o language model and an iterative validation pipeline.

The system successfully generates and validates senders for all 223 event–version combinations currently defined in the Eiffel Protocol, achieving 100% schema coverage. All generated senders pass a multi-tier testing suite that includes dynamic validation, handwritten black-box tests, failure-mode tests, and Rust-side integration tests with RabbitMQ publishing verification. The final product is a distributable release artifact containing a user-facing CLI tool with both an interactive guided mode and a passthrough mode for automation. The contribution of the thesis can be understood on two levels. On the engineering level, the project delivers a working Linux-based Eiffel sender toolchain and release artifact. On the scientific level, it contributes observations about the conditions under which LLM-assisted, schema-driven code generation can be made reliable, and about the limitations introduced by non-determinism, model dependency, and schema variation.

Regarding the research questions:

RQ1: A schema-driven pipeline with structured prompt construction, LLM-based generation, and iterative validation can produce correct Eiffel event senders for all event types and schema versions. The schema-driven design helps the system scale to new events and versions by using the Eiffel schemas as the primary structural source of truth. However, this scalability also depends on supporting helper logic for compatibility handling, prompt construction, and generated sender integration.

RQ2: Correctness can be systematically validated through a four-stage pipeline (compilation, execution, JSON parsing, schema validation) combined with multi-tier testing. The layered approach catches different categories of errors at different stages, and structured error feedback enables automated recovery from generation failures.

RQ3: LLM-based code generation offers greater flexibility than template-based approaches for producing complete, runnable programs from diverse schemas, but introduces non-determinism that requires validation and retry mechanisms. The quality of the generated output is highly sensitive to prompt design, and significant prompt engineering effort was required to achieve reliable first-attempt generation.

The thesis demonstrates that AI-assisted code generation, when combined with rigorous automated validation and a schema-driven design, is a viable approach for producing schema-compliant and practically validated tools at scale in a domain with a formally specified protocol. The complete CI/CD automation, including scheduled schema synchronization with automatic sender generation, supports adaptation to protocol evolution with limited manual effort, provided that new schema patterns remain compatible with the existing helper logic. The work was also shaped by the industrial project setting, in which the development of an AI-assisted Linux sender generation approach was part of the original assignment provided in collaboration with Nexer. Although the implemented results show that LLM-based schema-driven generation was viable in this project setting, no deterministic template-based baseline was implemented, and the thesis therefore does not empirically establish superiority over alternative non-LLM approaches.

Bibliography

[1] Eiffel Community, "Eiffel Protocol," 2024. [Online]. Available: https://eiffel-community.github.io/. [Accessed: May 15, 2026].

[2] Eiffel Community, "About the Eiffel Community," 2024. [Online]. Available: https://eiffel-community.github.io/community.html. [Accessed: May 15, 2026].

[3] Eiffel Community, "eiffel: Eiffel Protocol vocabulary, schemas, and samples," GitHub repository, 2024. [Online]. Available: https://github.com/eiffel-community/eiffel. [Accessed: May 15, 2026].

[4] A. Behrami and L. Agnezon, "Eiffel Sender: Python-based CLI tool for generating Eiffel event senders," unpublished course project, Chalmers University of Technology, 2025.

[5] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema: A media type for describing JSON documents," Internet Engineering Task Force, Internet-Draft draft-bhutton-json-schema-01, Jun. 2022. [Online]. Available: https://json-schema.org/specification. [Accessed: May 15, 2026].

[6] OpenAPI Initiative, "OpenAPI Specification," version 3.1.0, 2021. [Online]. Available: https://spec.openapis.org/oas/latest.html. [Accessed: May 15, 2026].

[7] OpenAPI Generator Contributors, "OpenAPI Generator," 2024. [Online]. Available: https://openapi-generator.tech/. [Accessed: May 15, 2026].

[8] Google, "Protocol Buffers: Language-neutral, platform-neutral extensible mechanisms for serializing structured data," 2024. [Online]. Available: https://protobuf.dev/. [Accessed: May 15, 2026].

[9] Quicktype Contributors, "quicktype: Generate types and converters from JSON, Schema, and GraphQL," 2024. [Online]. Available: https://quicktype.io/. [Accessed: May 15, 2026].

[10] W. X. Zhao *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023. [Online]. Available: https://arxiv.org/abs/2303.18223.

[11] OpenAI, "GPT-4o model documentation," 2024. [Online]. Available: https://platform.openai.com/docs/models. [Accessed: May 15, 2026].

[12] J. White *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023. [Online]. Available: https://arxiv.org/abs/2302.11382.

[13] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code

generation," in *Proc. 37th Advances in Neural Information Processing Systems (NeurIPS)*, 2023. [Online]. Available: https://arxiv.org/abs/2305.01210.

[14] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374.

[15] S. Chen, J. Peng, B. Ray, and M. Nagappan, "Teaching large language models to self-debug," in *Proc. 12th Int. Conf. on Learning Representations (ICLR)*, 2024. [Online]. Available: https://arxiv.org/abs/2304.05128.

[16] VMware, "RabbitMQ Documentation," 2024. [Online]. Available: https://www.rabbitmq.com/docs. [Accessed: May 15, 2026].

[17] RabbitMQ, "AMQP 0-9-1 Model Explained," 2024. [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts [Accessed: May 15, 2026].

[18] S. Klabnik and C. Nichols, *The Rust Programming Language*, 2nd ed. San Francisco, CA, USA: No Starch Press, 2023. [Online]. Available: https://doc.rust-lang.org/book/.

[19] Clap Contributors, "clap: Command Line Argument Parser for Rust," 2024. [Online]. Available: https://docs.rs/clap/latest/clap/. [Accessed: May 15, 2026].

[20] D. Tolnay, "serde_json: A JSON serialization library for Rust," 2024. [Online]. Available: https://docs.rs/serde_json/latest/serde_json/. [Accessed: May 15, 2026].

[21] Lapin Contributors, "lapin: AMQP client library for Rust," 2024. [Online]. Available: https://docs.rs/lapin/latest/lapin/. [Accessed: May 15, 2026].

[22] GitHub, "GitHub Actions Documentation," 2024. [Online]. Available: https://docs.github.com/en/actions. [Accessed: May 15, 2026].

A

Example Generated Sender

This appendix shows a simplified excerpt from a generated sender for `EiffelActivityCanceledEvent` version 5.0.1, illustrating the structure of the generated Rust code.

```
use clap::Parser;
use serde_json::{json, Map, Value};
use uuid::Uuid;
use chrono::Utc;

#[derive(Parser)]
#[command(name = "eiffelactivitycanceledevent-v5_0_1")]
struct Args {
    #[arg(long = "data-reason")]
    data_reason: String,

    #[arg(long = "meta-tags")]
    meta_tags: Option<Vec<String>>,

    #[arg(long = "links")]
    links: Option<Vec<String>>,

    #[arg(long = "send", default_value_t = false)]
    send: bool,

    #[arg(long = "rabbitmq-uri",
          default_value = "amqp://guest:guest@localhost:5672/%2f")]
    rabbitmq_uri: String,

    #[arg(long = "rabbitmq-queue",
          default_value = "eiffel-events")]
    rabbitmq_queue: String,
}

fn main() {
    let args = Args::parse();

    // Build meta section (automatic fields)
    let mut meta = Map::new();
    meta.insert("id".into(),
        Value::String(Uuid::new_v4().to_string()));
    meta.insert("type".into(),
        Value::String("EiffelActivityCanceledEvent".into()));
    meta.insert("version".into(),
        Value::String("5.0.1".into()));
    meta.insert("time".into(),
        Value::Number(Utc::now().timestamp_millis().into()));

    if let Some(ref tags) = args.meta_tags {
        let tag_values: Vec<Value> = tags.iter()
            .map(|t| Value::String(t.clone())).collect();
        meta.insert("tags".into(), Value::Array(tag_values));
    }

    // Build data section (from CLI arguments)
    let mut data = Map::new();
    data.insert("reason".into(),
        Value::String(args.data_reason.clone()));
```

```
    // Build links section
    let mut links_array = Vec::new();
    if let Some(ref link_list) = args.links {
        for link_str in link_list {
            let parts: Vec<&str> =
                link_str.splitn(2, '=').collect();
            if parts.len() == 2 {
                links_array.push(json!({
                    "type": parts[0],
                    "target": parts[1]
                }));
            }
        }
    }

    // Assemble complete event
    let mut event = Map::new();
    event.insert("meta".into(), Value::Object(meta));
    event.insert("data".into(), Value::Object(data));
    if !links_array.is_empty() {
        event.insert("links".into(),
            Value::Array(links_array));
    }

    let json_output = serde_json::to_string_pretty(
        &Value::Object(event)).unwrap();
    println!("{}", json_output);

    if args.send {
        // RabbitMQ publishing logic (injected)
        // ...
    }
}
```

B

Example LLM Prompt Excerpt

This appendix shows a simplified excerpt of the prompt constructed by `prompt_builder.py` for generating a sender. The actual prompt is longer and includes the full JSON Schema.

```
Generate a complete Rust CLI program that creates an
EiffelActivityCanceledEvent version 5.0.1 Eiffel event.

The program must:
- Use clap for argument parsing with derive macros.
- Accept required fields as long CLI flags named after
  their schema path (e.g., data.reason -> --data-reason).
- Automatically generate meta.id (UUID v4), meta.type,
  meta.version, and meta.time (UTC milliseconds).
- Build a JSON object with "meta", "data", and "links"
  top-level keys.
- Only include optional fields if provided via CLI.
  Do NOT include null values or empty objects.
- Parse links as TYPE=UUID pairs.
- Print the JSON to stdout using serde_json.

The JSON Schema for this event is:
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "meta": { ... },
    "data": {
      "type": "object",
      "properties": {
        "reason": { "type": "string" }
      },
      "required": ["reason"]
    },
    "links": { ... }
  },
  "required": ["meta", "data", "links"]
}

Do NOT:
- Generate placeholder or dummy values.
- Include empty objects or arrays when no data is given.
- Use unwrap() on user input without validation.
- Include any markdown formatting in your response.
```

C

Example Eiffel JSON Schema

This appendix shows a simplified version of the JSON Schema for `EiffelActivityCanceledEvent` version 5.0.1, illustrating the structure that drives the generation pipeline.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "meta": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string",
          "pattern": "^[0-9a-f]{8}-..."
        },
        "type": {
          "type": "string",
          "enum": ["EiffelActivityCanceledEvent"]
        },
        "version": {
          "type": "string",
          "enum": ["5.0.1"]
        },
        "time": {
          "type": "integer"
        },
        "tags": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "required": ["id", "type", "version", "time"],
      "additionalProperties": false
    },
    "data": {
      "type": "object",
      "properties": {
        "reason": { "type": "string" },
        "customData": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "key": { "type": "string" },
              "value": {}
            },
            "required": ["key", "value"]
          }
        }
      },
      "required": ["reason"],
      "additionalProperties": false
    },
    "links": {
      "type": "array",
      "items": {
        "type": "object",
```

```
        "properties": {
          "type": { "type": "string" },
          "target": { "type": "string" }
        },
        "required": ["type", "target"]
      }
    }
  },
  "required": ["meta", "data", "links"]
}
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY