CHALMERS

CHALMERS

NEXER

Schema driven AI-Assisted Generation of PowerShell Modules for Eiffel Events in CI/CD Pipelines

Developing and testing AI-Assisted PowerShell modules from JSON schemas for creating, validating and sending Eiffel Events

Bachelor's Thesis in Computer Engineering (TIDAL)

Muhammad Usman

Department of Computer Science & Engineering

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 20xx
www.chalmers.se

Degree project report 2026

Schema driven AI-Assisted Generation of PowerShell Modules for Eiffel Events in CI/CD Pipelines

Developing and testing AI-Assisted PowerShell modules from JSON schemas for creating, validating and sending Eiffel Events

Muhammad Usman

Academic Supervisor: Sakib Sistek
Department Examiner: John J. Camilleri
Industry Supervisors: Nikolai Dahlberg & Ankita Rahavachari

CHALMERS

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

Schema driven AI-Assisted Generation of PowerShell Modules for Eiffel Events in CI/CD Pipelines
Developing and testing AI-Assisted PowerShell modules from JSON schemas for creating, validating and sending Eiffel Events.

Muhammad Usman

Supervisor: Nikolai Dahlberg & Ankita Rahavachari, Nexer Group
Examiner: John J. Camilleri, Computer Science and Engineering

Degree project report 2026

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

Abstract

Modern CI/CD pipelines rely on structured event data to provide traceability, automation, and observability across the software development lifecycle. The Eiffel protocol addresses this need by formally specifying build, test, and deployment events through JSON Schemas, making it well-suited for schema-driven tooling. Despite this, practical integration of Eiffel events into CI/CD pipelines remains largely improvised, and no widely adopted approach exists for Windows-based environments. This thesis investigates how a schema-driven and AI-assisted approach can be used to automatically generate a Windows PowerShell module that creates, validates, and sends Eiffel events while remaining testable and maintainable.

A Python-based generator was developed that parses the official Eiffel event JSON Schemas and produces PowerShell modules exposing cmdlets for event creation, schema validation, and publication to RabbitMQ. An AI model, GPT-4.1, is integrated as a controlled build-time repair mechanism and is permitted to modify generator logic when system tests fail, but never allowed to change the human-written tests themselves.

The entire workflow of generation, testing, building, and releases is automated through GitHub Actions and published via GitHub Releases. The resulting product scales from a single event and version to the full Eiffel event registry by treating new events as new registry entries rather than as new code, with self-correcting validation gates that catch regressions automatically.

Acknowledgements

I would like to express my sincere gratitude to Nexer Group for providing the opportunity to carry out this thesis project and for offering both the technical challenge and the supportive environment that made this work possible. I am especially grateful to my industrial supervisors, Nikolai Dahlberg and Ankita Rahavachari, for the continuous guidance, technical insight, and valuable feedback throughout the project, and for taking the time to share their expertise on the Eiffel protocol and the industrial CI/CD practices. I would also like to thank my academic supervisor at Chalmers, Sakib Sistek, for the helpful discussions, constructive feedback, and support throughout the thesis work. Finally, I would like to thank my family and friends for their continuous encouragement and support during the course of this project.

Muhammad Usman, Gothenburg, May 2026

List of Acronyms

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

Acronym	Expansion
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery / Continuous Deployment
CLI	Command Line Interface
DLL	Dynamic-Link Library
GAV	Group, Artifact, Version (Maven coordinate triple)
GH	GitHub
JSON	JavaScript Object Notation
LLM	Large Language Model
MOTW	Mark-of-the-Web
MVP	Minimum Viable Product
PA	ParameterAttribute (PowerShell)
PR	Pull Request
URI	Uniform Resource Identifier

List of Figures

Figure 1 High level architecture diagram 8

Figure 2 Detailed development and GH Actions CI flow 9

Figure 3 Detailed user interaction flow chart 9

Figure 4 Simplified development flow 10

Figure 5 Simplified user flow 10

Figure 6 Local user workflow 15

Figure 7 GitHub Release MVP user workflow 15

List of Tables

Table 1 Mapping of validation methods to Chapter 3 results 10

Table 2 The scope and validation results 21

Table of Contents

List of Acronyms iii
List of Figures iv
List of Tables iv
1. Introduction 1
1.1 Background and Purpose 1
1.2 Approach 2
1.2.1 Requirements 2
1.2.2 MVP 2
1.2.3 Stretch Goals 3
1.2.4 Technologies Involved 3
1.2.5 Expected Outcome 3
1.3 Limitations 4
1.3.1 MVP Scope 4
1.3.2 Debugging and Maintenance Complexity 4
1.3.3 AI-Assisted Code Generation Reliability 4
1.3.4 PowerShell and JSON Schema Integration Challenges 4
1.3.5 CI/CD and Testing Risks 5
1.4 Evaluation Plan 5
1.5 Validation Methods 6
2 Method 6
2.1 High-Level Strategy 6
2.2 Software Engineering Practices 6
2.3 System Architecture 6
2.4 Design of implementation 9
3 Results 10
3.1 Overview 10
3.2 Schema-Driven Generation Pipeline 11
3.3 Runtime Validation 12
3.4 RabbitMQ Integration 12
3.5 Testing 13
3.6 Continuous Integration and Release 14
3.7 End-to-End Runtime Flow & MVP Fulfilment 14
3.8 Stretch Goals Stage 1: Event-Level Facade Architecture 16
3.8.1 Stage 1: Why the Facade Design Pattern 16
3.8.2 Dynamic Parameter Forwarding 16
3.8.3 Tab Completion and Release Packaging 16
3.8.4 Behavior-Driven Testing 17

3.8.5 Outcome of Stage 1 17
3.9 Stretch Goal Stage 2: Multi-Event Architecture 17
3.9.1 Top-Level Façade 17
3.9.2 Event Registry and Version Policies 18
3.9.3 Adding Events Through One Registry Change 18
3.9.4 Verification Tier Strengthening 19
3.9.5 Outcome of Stage 2 19
3.10 Stretch Goal Stage 3: Full Catalogue and Generator Hardening 19
3.10.1 Reducing CI Cost Through Validated Caching 20
3.10.2 Pre-Flight Validation Contracts 20
3.10.3 Test Assumptions and Schema Coverage 21
3.10.4 Outcome of Stage 3 21
3.11 Stretch Goals Summary 21
3.12 Integrating the tool into a Build Pipeline 22
4. Conclusion 23
4.1 Reflection on the Approach 23
4.2 Contributions 24
4.3 Limitations 24
4.4 Reflection on Sustainability 25
4.5 Future Work 25
4.6 Closing Remarks 26
Bibliography 27
Appendix 28
Appendix A: Markdown code for prompt builder 28
Appendix B: Generated PowerShell module functionality 34
B.1: Runtime creation, validation, and sending of an EiffelArtifactPublishedEvent payload 34
B.2: Runtime creation, validation, and sending of an EiffelActivityFinishedEvent payload 34
B.3 RabbitMQ management view showing the published Eiffel event messages 34
Appendix C: EVENT_REGISTRY Configuration and Generated Module Structure 35
C.1: EVENT_REGISTRY configuration used by the generator 35
C.2: Representative generated module structure for EiffelArtifactPublishedEvent 36
Appendix D: Schema-Alignment Test Suite 37
D.1: SchemaAlignment.Tests.ps1, independent schema-alignment tests for generated Eiffel modules 37
Appendix E: CI Workflow and Reproducible Pipeline Definition 44
E.1: GitHub Actions CI workflow for generation, validation, packaging, and release 44
E.2: Scheduled schema-update workflow 50
Appendix F: Initial Project Timeline 52

F.1: Initial Gantt chart from the thesis planning report.....52

1. Introduction

1.1 Background and Purpose

Continuous Integration and Continuous Delivery/Deployment (CI/CD) is the practice of automatically building, testing, and releasing software each time a developer makes a change. Its defining property is reproducibility. Every push goes through the same automated pipeline, and the output for a given commit is deterministic. This is what makes a release trustworthy, since anyone can reproduce it from the source. Modern CI/CD pipelines depend on structured event data to provide traceability, automation, and observability across the software development lifecycle [1]. Without a standardized event vocabulary, every team that wishes to correlate builds, tests, and deployments across tools is forced to invent its own ad-hoc message format, which limits the reusability of CI/CD tooling across organizations.

The Eiffel Protocol addresses precisely this problem. It is an event-driven protocol used in CI/CD environments to describe and communicate events related to software development processes such as builds, tests, and deployments. The way it works is that every significant step in the pipeline emits an event describing what just happened. For example, when a build step finishes, it emits an EiffelArtifactCreatedEvent describing what was built, when tests finish, an EiffelTestCaseFinishedEvent, and when a release goes out, an EiffelArtifactPublishedEvent. Each Eiffel event type and version is formally specified using JSON Schemas that define the required fields, types, and constraints, making the protocol well-suited for automated, schema-driven tooling. The Eiffel Protocol was originally developed by Ericsson to standardize how its internal CI/CD systems communicated build and test events [2], and it is currently maintained as a community-driven initiative. Compared to alternatives such as CloudEvents, which is a generic event format without domain-specific structure, Eiffel is uniquely suited to the CI/CD domain because it specifies not only the format but also the semantic content of build, test, and deployment events, including the links between them. This makes traceability across the software development lifecycle a property of the protocol itself rather than something each consumer must reconstruct.

Despite the quality of the protocol, practical integration of Eiffel events into CI/CD pipelines is still largely improvised. There is no widely adopted industry standard for how to construct, validate, and publish Eiffel messages from inside a pipeline, and there is no specific readily available tooling for Windows-based CI/CD environments where PowerShell [3] is the dominant platform, this is the gap the thesis addresses.

The proposed solution is a Windows PowerShell-based tool that generates Eiffel message senders directly from the official JSON Schemas, supporting all defined Eiffel event types and versions. Establishing a schema-driven approach to message sender generation, enabling consistent and maintainable integration of Eiffel events. The events themselves are intended to flow through a message broker, a component that sits between publishers and consumers. RabbitMQ [4] is selected as the message broker because it is an established reference broker for Eiffel deployments and supports AMQP [5], the Advanced Message Queuing Protocol used by every existing Eiffel consumer in production. Extending this approach across platforms, such as Windows and Linux, and across tooling formats, such as PowerShell modules and Python CLI tools, contributes toward a unified and reusable solution for Eiffel message integration in CI/CD pipelines.

Recent advances in AI-assisted code generation [6] enable new approaches to automatically generating and maintaining software directly from formal specifications. When combined with strong testing and CI/CD practices, AI can be used not as a replacement for software engineering but as a controlled mechanism for maintaining correctness and reducing manual effort. The thesis specifically uses GPT-4.1 (see Section 2.2) as a build-time repair mechanism whose output is gated by human-written tests and schema validation. The concrete problem this thesis aims to solve is therefore:

How can a schema-driven and AI-assisted approach be used to automatically generate a Windows PowerShell module that correctly creates, validates, and sends all versions of Eiffel events, while remaining testable, maintainable, and suitable for industrial CI/CD usage?

Solving this problem is relevant both academically, by exploring AI-assisted and specification-driven software generation, and industrially, by providing Nexer Group with a reusable and extensible solution for integrating Eiffel events into Windows-based CI/CD pipelines.

1.2 Approach

This section describes the approach used to realize the aim, including the implementation requirements, the MVP scope, the stretch goals, the technologies involved, and the expected outcome. The detailed system architecture and design are presented in Section 2 (Method).

1.2.1 Requirements

The following requirements were agreed with Nexer at the start of the project and constrain how the aim is realized: The generated tool must be automatically tested with system tests; when a test fails, it should be sent to an AI that tries to fix the generator without changing the tests. Tests for the generated tool must be written by humans rather than generated, to ensure that the tools follow the requirements rather than the other way around. All changes to code must be handled through pull requests. The generated tool must be built and released inside the Git repository using the GitHub Releases functionality, and GitHub Actions must be used for running automatic tests and deliveries. Finally, a README and documentation about how to run the software must be present in the GitHub repository.

1.2.2 MVP

Given the scope and complexity of the project, it is important to clearly define a Minimum Viable Product (MVP) to ensure that the core objectives of the thesis can be achieved within the available timeframe. The MVP focuses on validating the fundamental technical implementation and reducing project risk early on, while additional functionality and scaling are handled through stretch goals.

The MVP of this thesis is a schema-driven, automatically generated Windows PowerShell module that supports at least one Eiffel event type and one schema version, can be generated automatically from the official Eiffel JSON Schema, and provides PowerShell cmdlets to create an Eiffel event payload, validate the payload against the JSON Schema, and send the validated event to RabbitMQ. It includes human-written system tests verifying both schema validation correctness and successful event publication to RabbitMQ, is built and tested automatically using GitHub Actions, and is released as a downloadable artifact using GitHub Releases.

An early proof of concept was implemented where a simple PowerShell module was generated and used to send an Eiffel event with a relatively simple schema structure to a RabbitMQ instance. This proof of concept validated the end-to-end workflow, schema handling, PowerShell execution, messaging infrastructure, and CI integration before extending the solution to support the full project scope. The MVP was deliberately built focusing on a simple JSON schema to ensure the focus remained on the correct implementation of the architecture and workflow, and to avoid early friction from more complex schema structures.

1.2.3 Stretch Goals

Once the MVP was complete and functional, the following stretch goals were pursued. Support for all Eiffel event types and schema versions, extended AI-assisted repair capabilities for more complex generation failures, more advanced schema handling, including better error messages for invalid payloads, and improved PowerShell user experience, such as clearer parameter validation and help documentation. The first three of these were realized in successive post-MVP stages and are described in detail in Sections 3.8 through 3.10.

From a technical perspective, successful completion of the MVP means a fully automated end-to-end execution in which a generated PowerShell module is used to create, validate, and send an Eiffel event to a RabbitMQ instance. Successful communication is confirmed by the event being published to RabbitMQ without validation errors and being observable in a bound queue. Additionally, all system and integration tests must pass in GitHub Actions, demonstrating that the workflow is reproducible and stable in a CI environment, before being published via GitHub Releases.

1.2.4 Technologies Involved

The toolchain uses Python for the PowerShell module generator, PowerShell 7 (pwsh) as the runtime for the generated modules, and GPT-4.1 as the AI model for generator-side code repair. Continuous integration and delivery are implemented in GitHub Actions, with distribution through GitHub Releases. Pytest is used as the testing framework for the Python generator and is integrated into GitHub Actions. Pester is used for system and integration tests of the generated PowerShell modules. NJsonSchema is used for runtime JSON Schema validation inside the PowerShell module via a thin .NET wrapper, and RabbitMQ is deployed locally via Docker Desktop for development and end-to-end testing.

1.2.5 Expected Outcome

The expected outcome of this thesis is a fully automated, schema-driven program that generates a Windows PowerShell module capable of reliably producing and sending all Eiffel events and that fulfills the requirements defined by Nexer for testing, automation, and delivery. The focus is on ensuring that the generated tool can be reliably tested, maintained, and released using modern software engineering practices. The project demonstrates how AI-assisted code generation can be safely integrated into modern software engineering workflows when combined with strict testing, CI/CD automation, and clear responsibility boundaries.

The resulting solution provides Nexer Engineering with a reusable foundation for Eiffel event integration on Windows platforms and contributes knowledge on specification-driven, AI-assisted software generation in an industrial context

1.3 Limitations

This section distinguishes between two categories of constraint. MVP scope refers to deliberate decisions to defer features in order to fit the project timeline. These are not failures of the approach but rather intentional choices about where to draw the line for the headline thesis claims. Real limitations refer to genuine constraints that affect the toolchain regardless of scope. Both categories informed early risk mitigation and provide the basis for reflection in Section 4.

1.3.1 MVP Scope

Supporting all Eiffel event types and versions significantly increases the generation cost and CI runtime. The MVP, therefore, is scoped to a single event and version, with the full registry handled through stretch goals (see Section 3.8 onwards). At the time of writing, all the currently existing 24 Eiffel event types in scope have been generated with full version coverage, but exhaustive coverage of every historical Eiffel event remains an operational decision rather than an architectural one.

The MVP-stage system tests mock the RabbitMQ publish call for determinism in CI. A broker-backed system test stage would strengthen the guarantees, but was not prioritized in favor of completing the end-to-end MVP within the planned timeframe.

1.3.2 Debugging and Maintenance Complexity

The combination of Python, PowerShell, AI services, and CI/CD pipelines makes testing and debugging more complex. To manage this, responsibilities are clearly separated between the generator and runtime module, extensive logging and test output are used to aid debugging, and all changes are handled via pull requests to maintain traceability.

1.3.3 AI-Assisted Code Generation Reliability

A genuine limitation is that AI-assisted generation or repair can produce incorrect or unstable code [7]. This risk is mitigated by restricting AI usage to build-time only, enforcing strict validation using JSON Schema [8], using human-written system tests as the correctness criterion [9], and limiting the number of AI retry attempts to a configurable maximum (currently five) to prevent unbounded behavior. If AI-assisted repair proves unreliable in certain cases, manual adjustments to the generator template can be made while keeping the AI-assisted mechanism as a supporting tool. The key reflection, discussed further in Section 4, is that AI compliance with prompt instructions comes at a cost. Each recurring AI failure mode had to be addressed through stricter prompt rules, stricter validator gates, or both. This effort represents a real maintenance cost that increases as the generated codebase grows.

1.3.4 PowerShell and JSON Schema Integration Challenges

Integrating JSON Schema validation into a PowerShell module using .NET libraries introduced potential problems related to schema complexity and library behavior. The Eiffel JSON Schemas use strict validation rules, and differences between JSON Schema implementations can lead to unexpected validation behavior or unclear error messages at runtime. To mitigate these risks, an early proof of concept was implemented to validate that the selected .NET JSON Schema library correctly supports the required Eiffel schema features. The schema validation logic was clearly isolated within the PowerShell module to allow adjustment or

replacement if limitations were identified, and validation behavior was thoroughly tested in CI using human-written system tests. The concrete outcome of these checks is reported in Section 3.4 (Runtime Validation).

1.3.5 CI/CD and Testing Risks

The project relies on automated testing and CI/CD pipelines to ensure correctness and reproducibility. While this approach strengthens software quality, it also introduces potential risks related to infrastructure and test complexity. Mitigation strategies include deploying RabbitMQ via Docker Desktop to provide a reproducible and isolated test environment, designing tests to fail fast and produce clear and actionable outputs, and stabilizing and validating the CI pipeline early in the project before expanding tests.

1.4 Evaluation Plan

The following questions will guide the evaluation:

1. **Correctness**
    - Can the generated PowerShell module successfully create, validate, and send all supported Eiffel event types and versions?
    - Do all system tests pass in CI?
2. **Maintainability**
    - Can updates to Eiffel JSON Schemas be handled by re-running the generator without manual PowerShell code changes?
    - Is the generator structure sufficiently modular to support future extensions?
3. **AI-Assisted Repair Effectiveness**
    - When tests fail, can the AI-assisted mechanism resolve the failures without modifying test cases?
    - Does this process preserve correctness and reproducibility?
4. **CI/CD Integration**
    - Are testing, building, and releasing fully automated?
    - Can the PowerShell module be reliably reproduced from the repository?
5. **Usability**
    - Can a Windows developer unfamiliar with the code use the generated module based solely on the documentation?
    - Are cmdlet interfaces intuitive and consistent with PowerShell conventions?

1.5 Validation Methods

Validation is performed through execution of automated system tests in GitHub Actions, where generated modules are verified through the full CI pipeline, including Pytest, generation, Pester, and release jobs. Additional validation is carried out through end-to-end testing by sending Eiffel events to RabbitMQ, manual inspection of the generated PowerShell module structure, and review and feedback from Nexer supervisors regarding industrial applicability.

2 Method

2.1 High-Level Strategy

The thesis builds upon an existing Python-based Eiffel event generator and extends it to support Windows PowerShell as the target platform. Instead of rewriting the generator, Python remains as the programming and generation language, while PowerShell will be the primary runtime artifact produced by the system.

2.2 Software Engineering Practices

To ensure industrial relevance and correctness, the practices enforced in this project follow established principles of continuous delivery, namely that every change should be testable, releasable, and traceable through an automated pipeline [10].

- Human-written system tests verifying the required behavior
- Pull-request (PR) only development workflow
- Automated CI/CD pipelines using GitHub Actions
- Automatic execution of tests on each Push and PR
- Automatic build and release of the PowerShell module using GitHub Releases
- Comprehensive documentation and README explaining installation and usage

The Python generator is integrated with the existing CLI tool, allowing users to select whether they want to generate a Python CLI tool or a PowerShell module.

2.3 System Architecture

The system is structured into three main layers:

1. Schema Input Layer

- Official Eiffel JSON Schemas act as the single source of truth.
- All Eiffel event types and all published schema versions must be supported.

2. Generator Layer (Python + AI)

The generator is implemented in Python to leverage existing tooling and ecosystem support. The Eiffel JSON Schemas are parsed and analyzed to extract information about:

- Event types
- Schema versions
- Required and optional fields
- Structural and validation constraints

Based on the extracted information, source code for a PowerShell module is generated, and an AI model (GPT-4.1) is integrated as a controlled repair mechanism. AI is invoked when system tests fail, and is allowed to modify generator logic but never allowed to modify test cases.

3. Generated Tool Layer (PowerShell Module)

- A Windows PowerShell module (.psm1) is generated, providing cmdlets to:
  - Create Eiffel events
  - Validate events against specific schema versions
  - Send validated events to RabbitMQ

Users can select event type and version via PowerShell parameters, and manually input the payload values based on the required Data and Links fields in the selected event versions JSON Schema.

As the project progressed, the Generated Tool Layer evolved from a single per-event-version module into a three-layer facade architecture (one top-level facade, one per-event facade, and per-version modules underneath). This evolution is a stretch-goal contribution and is described in Section 3.8.

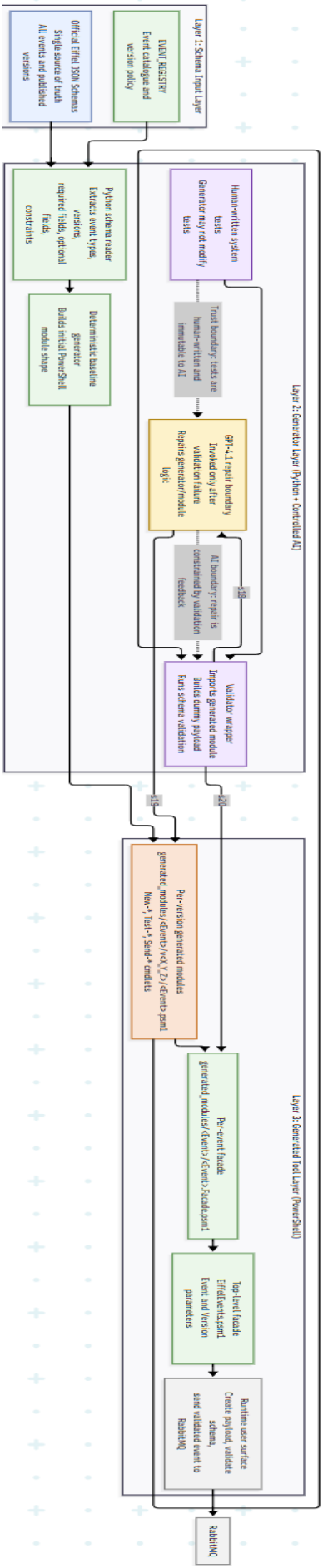


Figure 1 High level architecture diagram

2.4 Design of implementation

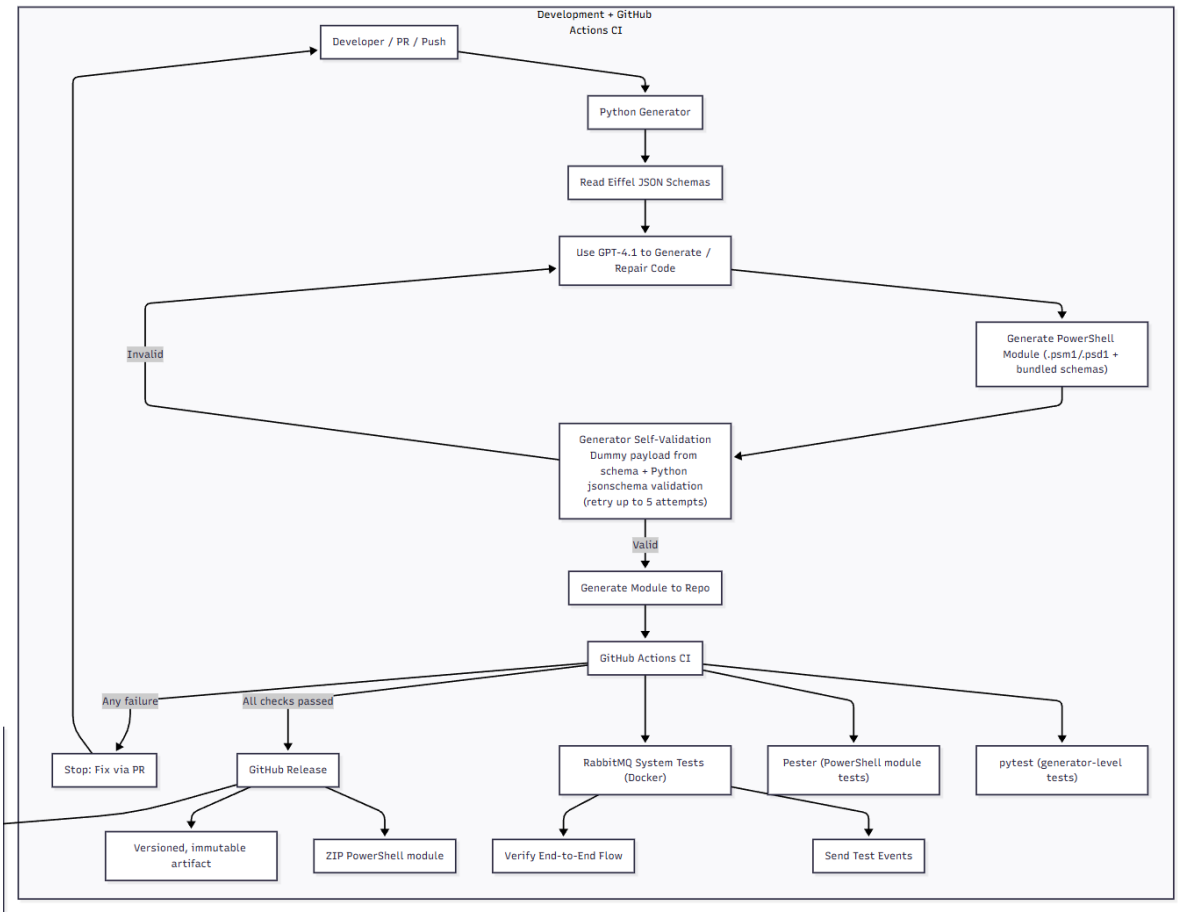


Figure 2 Detailed development and GH Actions CI flow

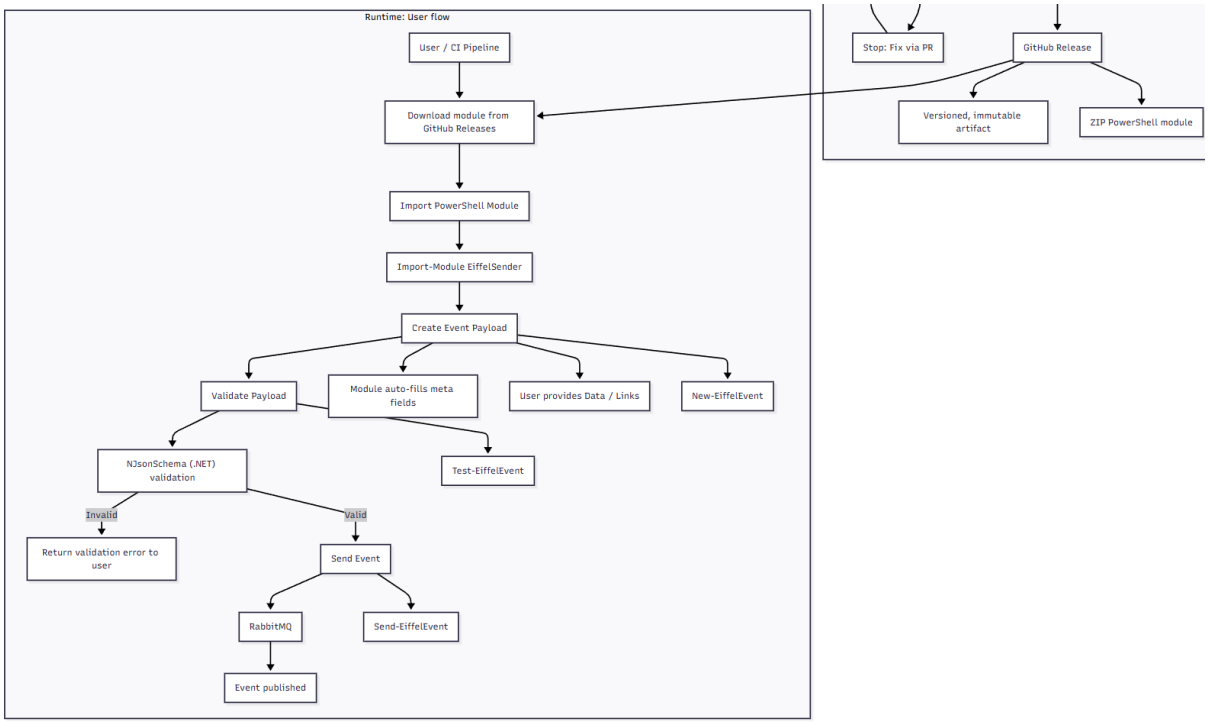


Figure 3 Detailed user interaction flow chart

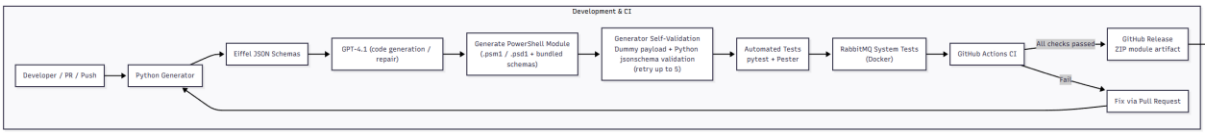


Figure 4 Simplified development flow

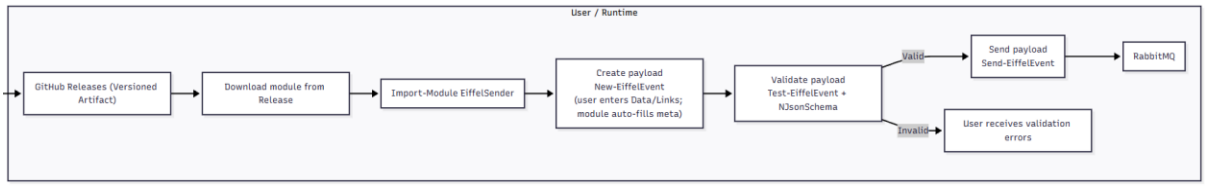


Figure 5 Simplified user flow

3 Results

3.1 Overview

The MVP was scoped to a single event type and version, EiffelArtifactPublishedEvent version 4.0.1, in order to validate the end-to-end architecture before scaling. By intentionally choosing this event, the core schema features (required fields, enum constraints, nested meta structure) were exercised without introducing the complexity of the full Eiffel schema catalogue, which proved sufficient to validate the architecture end-to-end. This confirmed empirically that the chosen .NET validator (NJsonSchema) handled the schema features Eiffel actually uses, that PowerShell 7 could host the validator without runtime mismatches, and that the AI-driven generation loop converged on a passing module within the configured retry attempts, three concrete prerequisites for later scaling the implementation that followed in Sections 3.8 to 3.10.

The MVP phase of the thesis produced a working, schema-driven implementation that automatically generates a Windows PowerShell module for a single Eiffel event type and version, validates event payloads against the official JSON Schema at runtime, and publishes validated events to RabbitMQ. The pipeline is validated by an automated test suite that runs on every push via GitHub Actions, with tested builds packaged and distributed through GitHub Releases. This section presents the resulting system, the implementation outcomes for each architectural layer defined in Section 2, and the extent to which the MVP requirements were fulfilled (Section 3.7), followed by the post-MVP stretch-goal stages that scaled the toolchain to the full Eiffel event registry (Sections 3.8 through 3.10).

Table 1 below maps the validation methods defined in Section 1.5 to the corresponding evidence presented in Chapter 3, clearly presenting the relationship between the planned evaluation strategy and the implementation results.

Validation method from Section 1.5	**Chapter 3 evidence**	**What it demonstrates**
Automated system tests in GH Actions	Sections 3.5, 3.6, 3.9.4, 3.10.4	Generated modules are verified through pytest, Pester, schema-alignment tests, and CI.
Full CI pipeline validation	Section 3.6	Generation, testing, artifact upload, packaging, and release are executed automatically.
End-to-end RabbitMQ testing	Sections 3.4, 3.7	Generated events can be validated and published through the intended messaging path.
Manual inspection of the generated PowerShell module	Sections 3.8, 3.9, 3.10	The generated module layout, facade layers, and registry-based scaling are inspectable.
Industrial applicability review and supervisor feedback	Sections 3.7, 3.8.5, 3.11	The final tool supports a realistic release and usage workflow suitable for industrial use.

Table 1 Mapping of validation methods to Chapter 3 results

3.2 Schema-Driven Generation Pipeline

The first architectural layer, schema parsing, was implemented as a Python component that fetches the official Eiffel JSON Schemas from their upstream source and stores them under a versioned local directory structure (/schemas/<EventName>/<Version>.json). This removed any hardcoded assumptions about event structure from the rest of the system and established the official schemas as the single source of truth for both code generation and runtime validation, as required.

Once the schemas have been fetched and committed under /schemas, the toolchain reads them locally rather than from the network on every generation run. This means that an upstream outage at generation time does not block the toolchain, it can continue to generate, validate, and release modules from the committed schemas. A re-fetch is required only when a new Eiffel schema version is published upstream, and that re-fetch is a deliberate, code-reviewable commit rather than an automatic pull. If the upstream source becomes permanently unavailable, the committed schema continues to function as a copy.

The generator layer was implemented in two stages. An initial deterministic generator (ps_module_creator.py) was built to produce a minimal PowerShell module directly from the parsed schema, without involving an AI model. This generator extracts required fields and enum values from the schema and emits corresponding PowerShell parameters with [Parameter(Mandatory)] and [ValidateSet(...)] attributes, producing a module that enforces schema constraints at the parameter level before any payload is even constructed. Deferring AI integration until after the deterministic path was validated kept early debugging simpler and confirmed that the generation-to-module pipeline was structurally valid.

Once the deterministic path was stable, the AI-assisted generation loop was introduced as a separate pipeline under tool_generator/ai_ps/. There are a few reasons for keeping the two pipelines separate rather than combining them. First, the deterministic generator served as a structural fallback, so if the AI pipeline failed unrecoverably for some schema, the deterministic output remained available as a known-correct baseline. Second, having two pipelines side-by-side made it possible to attribute regressions cleanly, and if a test failed only on the AI-generated module, the AI was responsible. If it failed on both, the cause lay in the schema parsing or template logic shared between them. This separation of concerns was particularly valuable during the runtime-validation debugging described in Section 3.3, where it was important to know whether failures were caused by the LLM or by the underlying validator integration.

The AI pipeline builds a strict prompt that specifies exported function names, required parameter shapes, import paths for shared runtime modules, and exact parameter names for the send cmdlet. The generated module output is written to a deterministic path, validated at runtime by importing the module and executing its generated cmdlets against sample data, and on failure, the structured error feedback is injected into the next generation attempt. The loop terminates on success or after a configurable maximum number of attempts.

If the loop exhausts its retry attempts without successfully producing a passing module, the most recent failing output is retained on disk at the deterministic generation path, and a non-zero exit code is returned by the runner. The retained output is intended for human inspection, and it is the most informative artifact for understanding why the loop failed. Additionally, it is overwritten on the next successful generation attempt and is not used for any downstream step, as the validator gate (Section 3.8.4) prevents a non-passing module from being packaged into a release, and the CI workflow surfaces the failure as a build error rather than continuing.

Several issues were encountered and resolved during the AI integration phase. Early generated modules used incorrect relative import paths for shared runtime modules, which were addressed by tightening the prompt contract with explicit path patterns. Generated send functions occasionally drifted to outdated parameter names, which were fixed by specifying exact parameter names in the prompt. The validator's auto-generated test data initially used a UUID format that did not match the schema's stricter pattern, which was corrected by

generating schema-compliant UUID values. The Send-* function's pre-send validation path was also found to forward empty optional parameters into the validator, tripping ValidateSet checks, and was corrected to pass only the payload during the validation step.

3.3 Runtime Validation

Runtime validation was one of the more technically challenging parts of the MVP. Given that the required payloads must be validated against the JSON Schema at runtime using a .NET library, NJsonSchema [11] was selected for this purpose.

The first attempt loaded NJsonSchema.dll directly into PowerShell using Add-Type, but this failed due to assembly resolution errors and ReflectionTypeLoadException issues. The root cause was identified as a runtime mismatch. Windows PowerShell 5.1 runs on .NET Framework 4.x, while the NJsonSchema build targeted .NET 6/8, resulting in System.Runtime version conflicts. The resolution was twofold. First, the project standardized on PowerShell 7 (pwsh), which runs on modern .NET and is compatible with the target assembly. This is best understood as a dependency requirement of the tool rather than a one-off technical fix since PowerShell 7 is a hard prerequisite for end users, documented as such in the README and enforced at module import time via a runtime check that throws a clear error if Windows PowerShell 5.1 is detected. The MVP scope deliberately does not target Windows PowerShell 5.1 because no clean technical path exists for hosting a .NET 6/8 assembly under a .NET Framework 4.x runtime without a parallel build of NJsonSchema for the older framework, which is out of scope for this thesis. Second, rather than invoking NJsonSchema directly from PowerShell, a dedicated .NET wrapper library, EiffelSchemaValidator was built, exposing a single static Validate(schemaJson, payloadJson) entry point. This wrapper is loaded into PowerShell via System.Reflection.Assembly::LoadFrom and invoked through a PowerShell module Validate-EiffelEventPayload.psm1, which becomes the single point of contact between PowerShell and the .NET validator.

An additional runtime bug was encountered in which PowerShell's pipeline object wrapping caused PSObject cannot be converted to System.String errors when passing JSON text to the validator. This was resolved by explicit casting of the schema and payload arguments to [string] before invocation. After these fixes, the validator reliably returned true for valid payloads and raised a clear exception for invalid ones, meeting the MVP acceptance criterion.

3.4 RabbitMQ Integration

The messaging layer was implemented using the RabbitMQ.Client .NET library, loaded into PowerShell in the same way as the validator. The integration exposed several compatibility issues that required architectural decisions rather than only code fixes.

The initial implementation used RabbitMQ.Client version 7.x, which had moved to an async-only API. Combining this with PowerShell's synchronous execution model led to handshake timeouts and unstable behavior that was difficult to debug. The project was therefore downgraded to version 6.6.0, which provides a stable synchronous API sufficient for the MVP's needs. The decision to go with 6.6.0 is an intentional constraint scoped to the MVP rather than a permanent constraint, as it was driven by the impedance mismatch between RabbitMQ.Client 7.x's async-only API and PowerShell's synchronous execution model, which is solvable but not within the MVP timeline. A natural next step beyond the MVP would be to migrate the sender to the 7.x async API, either by hosting the client inside an async runtime accessed from PowerShell, or by wrapping the async calls behind a synchronous .NET facade similar to the EiffelSchemaValidator wrapper. With the synchronous 6.6.0 API in place, the remaining issues were that ExchangeDeclare and BasicPublish required specific overloads, and the BasicPublish body parameter needed to be wrapped as System.ReadOnlyMemory[byte]. These were addressed once the correct 6.6.0 signatures were identified.

Connectivity and routing issues were resolved separately. The authentication issue in particular is an environment constraint scoped to the local MVP setup rather than a defect, as authentication initially failed because the default guest account is restricted to loopback connections, which is RabbitMQ's documented secure default. A dedicated RabbitMQ user was created via container environment variables to allow non-loopback access in test scenarios. Endpoint resolution was forced to IPv4 (127.0.0.1) to avoid IPv6 mismatches, and the cmdlet was refactored to accept explicit host/port parameters in addition to URI-based configuration. Finally, published messages were initially not visible in any queue because the target exchange had no matching queue binding, and RabbitMQ was silently dropping the messages. Creating a queue bound to the exchange with a routing key made the end-to-end flow observable, and from that point onward, messages produced by the generated module could be reliably verified in the RabbitMQ management UI.

The resulting Send-EiffelEventToRabbitMq cmdlet supports both URI-based and explicit-parameter connection configuration, rejects ambiguous mixed configurations, hides credentials in error output, and returns endpoint context when publishing fails. A higher-level helper script, Send-EiffelEventWithConfig.ps1, wraps validation and sending in a single call and is driven by a configuration file, removing the hardcoded values that were present in earlier iterations.

3.5 Testing

Testing was implemented as three coordinated layers, consistent with the thesis's requirement that human-written system tests verify the generated tool's behavior.

Unit tests were written in Pester to cover the individual PowerShell functions in isolation. The send cmdlet is tested for rejection of unsupported URI schemes, rejection of mixed URI and explicit connection parameters, and correct reporting of endpoint context on publish failures. The validator is tested for correct handling of missing schema files, rejection of non-hashtable payloads, and stability of its external call signature.

Integration tests were also written in Pester to exercise the full PowerShell-to-.NET validation path using fixed, known-good, and known-bad JSON "dummy" payloads. These tests decouple validation verification from the generator code by reading static sample files, and confirm that the validator enforces schema constraints through the real NJsonSchema path rather than a mock.

System workflow tests in Pester cover the end-to-end workflow with RabbitMQ publishing mocked. These tests verify that the workflow script validates payloads before sending, correctly passes URI configuration through to the send cmdlet, and falls back to host/port/vhost/credential parameters when no URI is provided. Mocking the publish call keeps the system tests deterministic and fast in CI while still exercising the full wiring.

In addition to the PowerShell test suites, **Python unit tests** in pytest cover the generator itself. The prompt builder is tested to ensure it produces the contracted function names and includes the required schema paths and enum rules. The module creator is tested with a mocked LLM response to verify that markdown code fences are stripped, that the output file is written to the expected path, and that retry feedback is correctly injected into subsequent prompts. Because these tests mock the LLM output, they run offline and are independent of model availability.

Finally, a **generated-module behaviour test suite** in Pester asserts that the AI-generated module exports the expected New-*, Test-*, and Send-* functions for the target event and version, that the payload produced by New-* passes validation through the shared .NET path, that -ValidateBeforeSend allows the publish path to be reached for valid payloads, and that it blocks publishing for invalid ones. These tests specify how the generated module should behave when used, without depending on how the AI produced it.

3.6 Continuous Integration and Release

The CI pipeline was implemented in GitHub Actions and went through several iterations before reaching a stable state. The final workflow is organized into three sequential jobs: a Python test job that runs pytest, a module-generation job that produces the generated PowerShell module and uploads it as a build artifact, and a Pester job that downloads the generated artifact and runs all PowerShell test suites against it. A fourth release job runs only after the test jobs succeed and is triggered by either a v* tag push or a manual workflow_dispatch.

Several CI-specific issues were encountered and resolved during this phase. The initial workflow mixed verification with source-control mutation by auto-committing generated senders back to the branch, which added unnecessary complexity and risked commit loops. This was replaced with an artifact-driven model in which generated modules are treated as build outputs rather than committed source. PowerShell test failures traced to $PSScriptRoot scoping were resolved by moving path resolution into BeforeAll blocks. A validator type-loading failure in CI was traced to the use of Windows PowerShell rather than PowerShell 7 and was resolved by switching the relevant steps to pwsh. Pester assertion syntax was updated from the legacy form to Pester 5 (Should -Be, Should -Match), and integration test helper scoping issues were addressed by loading sample payloads directly inside BeforeAll.

The release job is backed by a packaging script (Package-MvpRuntimeArtifact.ps1) that assembles the runtime artifact. The generated PowerShell module, the shared runtime modules (validator and sender), the pinned JSON schema, the configuration-driven helper script, the runtime DLLs from lib/, and a runtime README. The packaged zip is then published as a GitHub Release asset via the gh CLI. An early version of the packaging script bundled only a subset of the lib/ DLLs, which caused the runtime validator to fail on the user's machine with a missing NJsonSchema assembly error, and so this was fixed by copying the full set of runtime DLLs into the release package. With this in place, the release artifact is self-contained and can be downloaded and executed end-to-end from a new machine.

3.7 End-to-End Runtime Flow & MVP Fulfilment

The resulting MVP supports the following end-to-end flow from a user's perspective. The user downloads the release artifact from GitHub Releases, imports the generated PowerShell module, and invokes New-EiffelArtifactPublishedEventV4_0_1 with the required parameters: author identity, location type, location URI, link type, and link target. The returned payload object can be validated explicitly via Test-EiffelArtifactPublishedEventV4_0_1, or validated and sent in a single step via Send-EiffelArtifactPublishedEventV4_0_1 -ValidateBeforeSend -Uri <amqp-uri>. On success, the event is published to the configured RabbitMQ exchange and is observable in the bound queue. On validation failure, the send path is blocked, and a clear error is returned to the user before any network call is made.

Figure 6 Local user workflow

Figure 7 GitHub Release MVP user workflow

Looking at the MVP requirements defined in the planning report, the outcome is as follows. The system supports one Eiffel event type and one schema version, as planned, and the module is generated automatically from the official JSON Schema. The generated module provides cmdlets for creating, validating, and sending Eiffel events, and human-written system tests verify both schema validation correctness and successful publication to RabbitMQ. The build is tested automatically in GitHub Actions on each push, and tested builds are released as downloadable artifacts through GitHub Releases. The planning report's criterion for technical completion of a fully automated end-to-end execution in which a generated PowerShell module creates, validates, and sends an Eiffel event to a RabbitMQ instance, with all CI checks passing, has been met.

Several limitations of the MVP itself are worth stating explicitly, as they defined the starting point for the stretch-goal phase. The scope was deliberately restricted to a single event and version, and scaling to the full Eiffel catalogue had not been attempted. The AI generation loop's pass criterion was functional rather than a full quality gate. Pester checks were not yet integrated into the loop itself, and there are no categorized retry feedback buckets or semantic diff checks. The generated module relies on repository-relative import paths for shared runtime modules, which works for the MVP but will need to be generalized. Finally, the system tests mocked the RabbitMQ publish call for determinism in CI, however, a broker-backed system test stage would strengthen the guarantees but was not prioritized in favor of completing the end-to-end MVP within the planned timeframe.

3.8 Stretch Goals Stage 1: Event-Level Facade Architecture

With the MVP demonstrating end-to-end correctness for a single event and version, the first stretch goal was to scale to all schema versions of the MVP event without requiring users to know the exact version-tagged cmdlet name. This stage introduced a single public entry point per event, selectable by a -Version flag, while preserving the version-specific generated code that the AI generation loop produced.

3.8.1 Stage 1: Why the Facade Design Pattern

Before this stage, each version-specific module exported three version-tagged functions: New-EiffelArtifactPublishedEventV4_0_1, Test-...V4_0_1, Send-...V4_0_1. Users had to know, import, and reference the exact version-tagged cmdlet name. Switching versions required changing every place where the cmdlet was used.

The natural alternative would have been to generate a single mega-module that inlines every version. This was rejected because the AI generation loop produces one module per schema version, validates it, and retries on failure with feedback scoped to the broken version. Inlining all versions into one module would mean every regeneration risks breaking previously working versions and undoing the "regenerate only what changed" property that the version-specific layout was designed to preserve. The version-specific modules are therefore not leftover implementation artifacts, but a deliberate architectural choice to support the AI generation loop.

The facade design adds a hand-written facade module, <Event>.Facade.psm1, alongside the version-specific modules. The facade exports four functions: New-EiffelArtifactPublishedEvent -Version <x>, builds a payload for the requested version, Test-EiffelArtifactPublishedEvent -Version <x> -Payload $p validates a payload against the requested version's schema, Send-EiffelArtifactPublishedEvent -Version <x>, publishes to RabbitMQ, and finally Get-EiffelArtifactPublishedEventVersions enumerates available versions by reading the filesystem. The version-specific modules remain exported and callable directly, so the facade is additive rather than a replacement.

3.8.2 Dynamic Parameter Forwarding

The facade dispatches via PowerShell's DynamicParam block. When the caller invokes New-EiffelArtifactPublishedEvent -Version 4.0.1, PowerShell binds Version first, then DynamicParam fires. The facade resolves the version folder (v4_0_1/), lazily imports its module, inspects the underlying New-EiffelArtifactPublishedEventV4_0_1 cmdlet, and clones each underlying parameter name, type, and attributes such as ValidateSet and ValidatePattern into a RuntimeDefinedParameterDictionary. PowerShell then binds the remaining arguments against these dynamic parameters, and in the process, the facade passes on everything except Version to the underlying cmdlet.

A simpler hashtable API (-Data @{ ... }) was considered and rejected because it would have lost tab completion, typed parameters, per-schema ValidateSet/ValidatePattern contracts, and consistency with the existing per-version cmdlet usability. The trade-off is that facade complexity increases, but the user interface quality stays high.

3.8.3 Tab Completion and Release Packaging

ValidateSet was not used for the -Version parameter because its allowed values are fixed when the command is parsed. This means it would not automatically update if version folders were added or removed later. Instead, the module registers runtime ArgumentCompleter callbacks for -Version when it is loaded, so tab completion can reflect the versions currently available on disk.

A subtle but consequential issue surfaced at the release artifact stage. Windows tags every file inside a downloaded zip with a Mark-of-the-Web (MOTW) marker. Under the default RemoteSigned execution policy, PowerShell refuses to load unsigned .psm1 and .ps1 files with that marker. The facade would import successfully but then fail on lazy-loaded per-version modules with misleading "parameter not found" errors.

The mitigation documented in the README is a manually typed one-line command, Get-ChildItem -Recurse -File | Unblock-File, executed from inside the extracted release directory before importing the facade. This removes the Mark of the Web flag from the downloaded files. Because commands typed directly into the PowerShell prompt do not themselves carry Mark of the Web, the command can execute even when downloaded scripts are blocked. An equivalent helper script was avoided because it would also be downloaded and could be blocked during the first execution. Therefore, the README presents the manual command rather than a script-based workaround.

3.8.4 Behavior-Driven Testing

The Stage 1 test strategy emphasized behavior over implementation details. Five tiers of test were defined: event-level tests verifying that the facade routes New/Test/Send to each version, lists versions, rejects unknown versions, and that -ValidateBeforeSend blocks invalid payloads. Version-level tests to verify that each version-specific module exports the expected cmdlets and produces schema-valid payloads. Shared-runtime tests for the validator and sender modules. Finally, end-to-end integration test against a real schema and a system test through mocking RabbitMQ. Implementation-coupled tests that duplicated public behavior coverage were removed, leaving a smaller, faster suite with preserved behavioral coverage.

3.8.5 Outcome of Stage 1

Stage 1 delivered a single public API per event without breaking the version-specific abstraction required by the AI generation loop. The facade is additive, deterministic, and manually implemented. CI now produces a signed-off, MOTW-aware release zip (eiffel-<tag>.zip) on every tag, and the release README is sourced directly from the repository README so that documentation stays single-sourced. The architecture is ready to scale to multiple events in Stage 2 with incremental changes rather than structural redesign.

3.9 Stretch Goal Stage 2: Multi-Event Architecture

Stage 1 showed that one event could have the same public API across multiple versions. Stage 2 checked whether the same idea still worked when supporting multiple event types. The goal was to keep the same API style for different events, without writing a separate facade manually for each one, and without forcing all events to handle versions in the same way.

3.9.1 Top-Level Façade

A third layer above Stage 1's per-event facade was introduced. EiffelEvents.psm1 lives at the repository root and exports five functions: New-EiffelEvent -Event <X> -Version <Y>, which builds a payload for any event-version pair. Test-EiffelEvent -Event <X> -Version <Y> -Payload $p validates against any pair's schema. Send-EiffelEvent -Event <X> -Version <Y> publishes to RabbitMQ with optional -ValidateBeforeSend. Get-EiffelEvents enumerates event folders that have a facade, and finally, Get-EiffelEventVersions -Event <X> enumerates versions for a specific event.

An alternative was to make the top-level facade call the per-event facade first, and then have the per-event facade call the version-specific cmdlet. This would have added three layers of DynamicParam handling for every command. However, this approach was rejected because the per-event facade does not add any extra schema knowledge. Both facade layers would read and copy the same parameters from the same version-

specific cmdlet, so the extra layer would only add complexity without improving the design. Instead, the top-level facade imports the correct version-specific module directly from v<X_Y_Z>/<Event>.psm1.

The per-event facade is still kept, but only as an alternative public API for workflows that focus on a single event. It is not used as a dependency by the top-level facade. In other words, both facades sit beside each other on top of the version-specific modules, but they are not stacked on top of each other.

3.9.2 Event Registry and Version Policies

tool_generator/ai_ps/config.py was extended with an EVENT_REGISTRY. This is a Python dictionary where each event name is mapped to either "all" or "latest". The registry defines which events should be generated and how many versions of each event should be included. It is used by the generator runner, the CI workflow, and the release packager, making it the single source of truth for event selection.

An initial approach that discovered events directly from the folders on disk was rejected because the filesystem does not reliably show what should be included. It only shows what currently exists. For example, a half-generated or leftover event folder could accidentally be included in the release zip. Disk discovery also gave the registry no trusted list to check against when verifying that all required events were present. A separate YAML or TOML configuration file was also considered but rejected to avoid introducing another file format and parser. A Python dictionary was simpler because the rest of the generator configuration already uses Python.

Different events used different version policies. The MVP event stayed on "all" because Stage 1 had already tested all nine of its versions. The Stage 2 events were set to "latest" because the priority at that stage was to get the latest version of every event working first to demonstrate that the system could scale across event types without immediately generating every historical version. The reasoning was that once the generator worked for the latest version of an event, support for the remaining versions could be added later by changing the event's registry policy from "latest" to "all". This reduced LLM cost and CI runtime during Stage 2, while keeping the design scalable.

Generating all historical versions for every Stage 2 event at that point would have increased cost and runtime before the basic event-wide coverage had been proven. Later, when full coverage became the goal, the entire registry was changed to "all" and is discussed further in Section 3.10.

3.9.3 Adding Events Through One Registry Change

The main architectural result of Stage 2 is that a new event can be added by changing only one line in EVENT_REGISTRY. No other manual code changes are required. This works because each part of the system gets its information from either the registry or the generated module structure on disk. The generator runner reads the registry, the per-event facade generation is generic and runs once for each registry entry, the top-level facade discovers available events at runtime by scanning generated_modules/, and tab completion uses the same scan. The release packager also reads the registry, checks that all required events are present, and includes the generated modules it finds. Finally, the CI tests avoid hard-coded event names by iterating over generated_modules/ directly.

Adding more events still increases operational cost as each new event can require more LLM tokens, longer CI runtime, and more generated code that must pass validation. However, it does not increase the amount of manually written facade code, test code, or event-specific engineering work. That complexity is handled by the architecture. This property was validated across several batches, for example, adding four Activity events, five more events from Announcement, Artifact, and Composition events, and so on. All the events were added through a single config.py change, with no changes to the facades, tests, release packager, or CI workflow.

3.9.4 Verification Tier Strengthening

Stage 2 added six new test files. These tests raised the validation level compared with Stage 1. In Stage 1, the main question was whether the cmdlet accepted the parameters it declared. In Stage 2, the tests checked a stronger requirement, whether the cmdlet interface actually matched the JSON Schema, and whether the released zip could be imported successfully on a clean machine.

The most important new test file was SchemaAlignment.Tests.ps1. It uses the JSON Schema as the source of truth and checks three things for each event and version pair. First, it checks coverage, meaning every non-meta required field in the schema must receive a non-empty value in the generated payload. This catches generator bugs where required fields are accidentally skipped. Second, it checks round-trip behavior as each required string parameter is given a unique test value, and the test verifies that the value appears somewhere in the generated payload. This catches parameters that the cmdlet accepts but never writes into the output. Third, it checks enum alignment, so if the schema defines a required enum with multiple valid values, the cmdlet must expose a matching ValidateSet. This test catches cases where the schema and the PowerShell parameter validation drift apart.

The tests use their own schema-reading logic instead of reusing the generator's schema-reading logic. This was done intentionally. Otherwise, if both the generator and the tests used the same parser, a bug in that parser could appear in both places. The generator could produce incorrect code, and the tests might still pass because they made the same mistake. By using a separate logic in the tests, the generated output is checked more independently against the JSON Schema.

Another important test was Release-ArtifactSmoke.Tests.ps1, which builds the same zip file that the CI release job would produce, extracts it into a temporary directory, and imports the module from there. It also runs cmdlet invocations inside a fresh pwsh subprocess so that the result is not affected by modules already loaded in the parent PowerShell session. This test covers release-specific problems that normal unit or integration tests might miss, such as zip layout, Mark of the Web behavior, DLL loading from lib/, the expected install layout, and module import order.

3.9.5 Outcome of Stage 2

After Stage 2, the generated module can provide a uniform public API across both schema versions and event types. In other words, users can interact with different events and different versions through the same user interface. Additionally, the architecture can add new events without adding new manually written code, as new events are read by the existing generator, facade, packaging, and test structure.

This was confirmed by the event-by-version matrix tests, which passed for every generated event and version found on disk, and the Stage 2 implementation process showed that the design scaled as intended, i.e., separate batches of new events were added, and each batch only required a single change to the registry.

3.10 Stretch Goal Stage 3: Full Catalogue and Generator Hardening

The final stretch-goal stage scaled the registry to the full Eiffel event catalogue in scope (24 events with full version coverage, totaling 222 generated modules across the event-version matrix), and hardened the generator and validator against several classes of bug that only surfaced in the long tail of Eiffel schemas. This stage also eliminated the marginal LLM cost of CI runs.

3.10.1 Reducing CI Cost Through Validated Caching

A practical consequence of registry growth was that every CI run regenerated every module from scratch, burning OpenAI tokens proportional to the registry size. This was fixed by two coordinated changes. First, the generation runner was extended with a skip-if-validates short-circuit, meaning if a module already exists at the deterministic generation path and its existing output passes the validator, the generation step is skipped, and no LLM call is made. A --force flag was added for the rare case (template change, prompt rewrite) when full regeneration is genuinely intended. Second, the generated module folder was moved from .gitignore into version control, so that CI starts from a checkout that already contains every previously validated module.

This may look like the usual anti-pattern of committing build artifacts, but in this project, the trade-off is intentional, as three properties make the trade favorable here. First, the generated artifacts are expensive to recreate. Regeneration uses LLM calls to OpenAI, not just local CPU time. Second, the artifacts are checked before they are reused. They are deterministic for the same generator inputs, such as the prompt, template, schema, and model version. More importantly, the validation step rejects any cached module that no longer satisfies the required contract. Third, the generated modules are needed by users at runtime, and users of the released PowerShell module should not need an OpenAI key just to use the tool. For this reason, the committed generated folder acts like a CI cache rather than a normal source folder. If the validator becomes stricter, as it did in Stages 2 and 3, old cached modules that no longer pass validation are rejected and regenerated automatically. This keeps the cache correct without manual tracking. The practical benefit is significant. Without the cache, every CI run risks becoming slow, costly, and rate-limited by LLM calls. However, with the cache, the generation step usually becomes validation-only and takes significantly less time.

3.10.2 Pre-Flight Validation Contracts

At full registry scale, three types of LLM-generated errors appeared. These errors had previously been caught later in the test pipeline, but Stage 3 moved them earlier into the validator. This means invalid modules are now rejected before they can reach downstream tests or packaging.

The first issue was the ByPayload contract. Some generated Test-* and Send-* cmdlets placed their -Payload parameter outside a named parameter set. This confused PowerShell's parameter-set resolver. Consequently, commands such as Test-* -Payload $p could fail even though the payload was being passed correctly. The fix was to require exactly two parameter sets: ByPayload, where only -Payload is mandatory, and Create, where the creation-style parameters are mandatory. The validator now checks this directly, if Test-* or Send-* do not provide a parameter set where -Payload is the only mandatory parameter, the generated module is rejected.

The second issue was the non-zero parameter count contract. Some Eiffel schemas do not define a top-level required array, even though nested objects such as data and meta do define required fields. The generator originally stopped when the top-level required array was missing, so it failed to visit required fields inside data.*. In some cases, this led to generated New-* cmdlets with no parameters at all. The earlier validator did not catch this because calling a zero-parameter cmdlet with zero arguments still succeeds. The schema reader was updated so that, at the root level only, required child objects are still visited when they contain their own required fields. The validator now also checks that if the schema requires non-meta fields, the generated New-* cmdlet must expose at least one non-common parameter.

The third issue was the reserved-name collision guard. Some schema fields produced parameter names that collided with existing PowerShell parameters. For example, data.uri became -Uri, which conflicted with the AMQP connection URI parameter on Send-*. To prevent this, the prompt builder now keeps a reserved name list. When a generated parameter name would collide, it is prefixed with its parent path segment. For example, data.uri becomes -DataUri. This rule is deterministic and visible in the generated cmdlet interface.

Together, these fixes show an important pattern in LLM-driven code generation. It is not enough to handle generation mistakes later in tests or with defensive workarounds, but rather, if an LLM-generated design decision affects users, the required behavior should be made explicit in the prompt and enforced by the validator. This way, incorrect modules are rejected early, and correct modules are more likely to be generated on the first attempt. Once a rule is checked directly by the validator, older safety workarounds are often no longer needed. One example is Stage 3's removal of the earlier workaround that manually forwarded the creation-style parameters to the underlying cmdlet. This shows how the generator-driven codebase can become cleaner over time as hidden assumptions are turned into explicit validation rules.

3.10.3 Test Assumptions and Schema Coverage

One issue worth noting is that the negative validation test was passing for the wrong reason on some events. The test used this command: Send-* -Payload @{} -ValidateBeforeSend

It is assumed that an empty payload would always fail schema validation. However, that assumption was incorrect as some Eiffel events do not define required fields at the top level, so an empty hashtable can still pass validation. For those events, the test did not pass because validation rejected the payload, but rather because the next step of sending the event to RabbitMQ failed for connection-related reasons. The test was fixed by using a payload that is invalid for every Eiffel event:

@{ meta = @{ type = 'NotARealEiffelEventType' } }

This works because every Eiffel schema restricts meta.type to the correct event name. Therefore, a fake event type always causes a real schema validation failure. This problem had the same root cause as the issue in Section 3.10.2, i.e., the implementation assumed that every Eiffel schema had top-level required fields. That assumption affected the schema reader, the validator, and this test, so all three had to be corrected.

3.10.4 Outcome of Stage 3

The final test suite is green at 336 passing, 0 failing, 59 skipped (intentional skips when a schema has no multi-value required enums) across 12 test files and 395 discovered cases. The registry covers 24 events with full version coverage (222 generated modules). CI runs much faster when no inputs have changed, and rate-limit failures have not recurred. The toolchain now scales to the full Eiffel event catalogue as a generation-cost decision rather than an engineering decision. The results are summarised in Table 2 below.

Metric	**Result**	**Interpretation**
Supported Eiffel event types	24	The generator covers the full configured event registry.
Generated PowerShell modules	222	One module is generated for each supported event-version pair.
Passing automated tests	336	The implemented test suite completed successfully.

Table 2 The scope and validation results

3.11 Stretch Goals Summary

Looking back at the stretch goals defined in the planning stage, every goal except generator support for output formats beyond PowerShell has been achieved. Support for all Eiffel event types and schema versions in scope is in place (Sections 3.8–3.10). Extended AI-assisted repair capabilities for more complex generation failures are in place via the validator contracts of Section 3.10.2. More advanced schema handling, including pre-flight rejection of incorrect output, is in place via the schema-alignment tier of Section 3.9.4. Improved PowerShell user experience, runtime tab completion, structured error output, MOTW guidance, and a single uniform API

across the full event catalogue, is in place. The generator-output-format extension is the natural next direction beyond the thesis.

3.12 Integrating the tool into a Build Pipeline

Having described the full toolchain across the MVP and the three stretch-goal stages, this section walks through a simple example of how the resulting PowerShell module is meant to be used in a Windows-based CI/CD environment. The example is illustrative rather than a deployed case, but it reflects the kind of integration the tool was designed to support.

Imagine a team at Nexer that builds a Windows installer for an internal product. Each time a developer pushes code, a CI/CD pipeline runs that compiles the installer, tests it, and makes it available for download. The team wants other parts of the organisation, such as dashboards and deployment tools, to know when a new installer has been published. Rather than each of those tools asking the build pipeline directly, the build pipeline can simply announce the event, and any interested tool can listen for it. The generated PowerShell module is what makes that announcement possible by running one short step at the end of the build pipeline.

After the installer has been built, the pipeline imports the top-level facade (Section 3.10.1) and calls its cmdlets in sequence. The CI/CD pipeline runs in a Windows environment, where the generated PowerShell module is imported and used to create, validate, and send the Eiffel message. The message describes the build result, while the schema validator, bundled with the module, checks the message against the official Eiffel JSON Schema before anything is sent over the network, and RabbitMQ acts as the message broker described in Section 1.1. The module sends the announcement to RabbitMQ, and RabbitMQ delivers it to any tool that has subscribed.

Two Eiffel events naturally appear in this scenario. Earlier in the pipeline, when the installer is first built, an EiffelArtifactCreatedEvent is emitted to say that the installer now exists. Then, when the installer is made available for download, an EiffelArtifactPublishedEvent is emitted to say where it can be found, with a link back to the creation event so consumers can trace the published file to the build that produced it. A downstream deployment tool, listening on RabbitMQ, can then pick up the published event and act on it, for example by fetching the installer and rolling it out to a test environment. The chain of events forms the kind of end-to-end traceability that the Eiffel Protocol is designed to provide and that the generated PowerShell module now makes accessible to Windows-based teams.

4. Conclusion

This thesis set out to investigate how a schema-driven and AI-assisted approach can be used to automatically generate a Windows PowerShell module that correctly creates, validates, and sends Eiffel events, while remaining testable, maintainable, and suitable for industrial CI/CD usage. The resulting toolchain answers that question affirmatively. A Python-based generator reads the official Eiffel JSON Schemas and emits PowerShell modules whose cmdlets construct, validate, and publish events to RabbitMQ. GPT-4.1 is integrated as a controlled, build-time repair mechanism whose output is gated by human-written tests and runtime schema validation. The full workflow of generation, testing, building, and release is automated through GitHub Actions and distributed via GitHub Releases. From the MVP, which targeted a single event and version to validate the end-to-end architecture, the toolchain was extended through three successive stretch-goal stages that scaled to the full Eiffel event catalogue in scope, totaling 24 events with full version coverage and approximately 222 generated modules.

4.1 Reflection on the Approach

A central concern throughout the project was that AI-assisted code generation should not be treated as a substitute for software engineering, but as a mechanism whose correctness is established by the same engineering disciplines that govern the rest of the system. Several deliberate architectural choices shaped how that principle was realized, and reflecting on them clarifies what the thesis contributes beyond the technical artifact itself. The decision to make the official Eiffel JSON Schemas the single source of truth aligned the generator, the runtime validator, and the test suite around one common anchor. Doing so had practical consequences that became increasingly important as the registry grew. Schema changes flow predictably through the toolchain, the validator catches drift between the generator and the schema before it reaches users, and tests use their own schema-reading logic rather than the generator's, so a bug in schema parsing cannot pass through both code and tests undetected.

The separation of the generator into a deterministic baseline and a parallel AI-assisted pipeline was made early and proved consequential later. Although the AI pipeline became the primary path, the deterministic generator remained available as a known-correct structural fallback and, more importantly, as a diagnostic tool. When system tests failed, the side-by-side pipelines made it possible to attribute regressions cleanly. A failure that appeared only on the AI-generated module pointed at the LLM, while a failure that appeared on both pointed at the schema parsing or shared template logic. This separation of concerns kept debugging traceable during the runtime-validation work in Section 3.4 and during the generator-hardening work in Section 3.10.

The facade pattern introduced in Stage 1 and extended in Stage 2 reflects the same principle of separating engineering concerns from generation strategy. The natural alternative, being a single large module that embedded every event and version, was rejected because it would have undone the AI generation loop's "regenerate only what changed" property. Instead, the version-specific modules remain the unit of generation, and the public interface that users interact with is provided by manually implemented facade modules layered above them. The result is that the user interface is uniform across events and versions, while the regeneration unit remains small enough for the AI loop to validate, retry, and fail safely. The use of DynamicParam rather than a hashtable API, and runtime ArgumentCompleter callbacks rather than static ValidateSet, were further deliberate choices that traded facade complexity for user experience and adaptability to filesystem state.

The handling of recurring AI failures became more consistent across the three stretch-goal stages. Rather than absorbing each failure with a defensive workaround at the consumption site, recurring problems were addressed by tightening the llm-prompt and the validator gate together. The pre-flight requirements introduced in Section 3.10.2, the parameter-set requirement, the non-zero parameter count rule, and the reserved-name collision guard are concrete examples of this approach. Each rule was first observed as a downstream test failure, then encoded both as an explicit instruction in the prompt and as a check in the validator. Once a rule

became enforceable, the corresponding workaround for the consumer was retired. This co-evolution of prompt and validator captures a more general lesson worth surfacing as a contribution. Compliance with prompt instructions has a cost that scales with the size of the generated codebase, and the maintenance burden lives in the prompt and validator rather than in the generated artifacts themselves. This is consistent with prior findings that LLM-generated code exhibits limited robustness to changes in input specifications and can produce subtly incorrect outputs even when prompts appear clear [7]. Treating AI as engineered infrastructure means investing in those two surfaces, rather than patching their outputs after the fact.

Several technical decisions were deliberately scoped rather than treated as defects. The standardization on PowerShell 7 was framed as a documented dependency of the tool rather than a workaround for a Windows PowerShell 5.1 limitation and is enforced at module import. The downgrade to RabbitMQ.Client 6.6.0 was acknowledged as an MVP-scoped trade-off driven by the impedance mismatch between the 7.x async API and PowerShell's synchronous execution model, with a clear migration path identified for future work. The committing of generated modules to version control, which would normally be an anti-pattern, was justified on three explicit grounds, namely the artifacts are expensive to recreate because regeneration consumes LLM tokens, the validator-as-cache-gate rejects any cached module that no longer satisfies the current contract, and end users of the released module should not require an OpenAI API key to consume the tool. These decisions are presented honestly as trade-offs, which is itself part of the thesis's reflective stance on AI-assisted software engineering.

Finally, the test architecture was designed to mirror the generation architecture. Behavior-driven tiers separate event-level concerns from version-level concerns and from shared-runtime concerns. The schema-alignment tests use independent schema-reading logic to prevent co-bugs with the generator, and the release-artifact smoke test exercises the same packaging path that real users invoke. The structure of the test suite is therefore an intentional part of the thesis contribution, not just a supporting detail.

4.2 Contributions

The thesis contributes a working, schema-driven, AI-assisted toolchain for generating, testing, and distributing PowerShell modules for the full Eiffel event catalogue in scope. The artifact is reproducible from the repository, validated automatically by GitHub Actions, and released as a self-contained zip file for industrial use within Nexer's Windows-based CI/CD environments. Beyond the artifact, the thesis contributes a documented pattern for incorporating AI as a controlled build-time mechanism, including the validator-prompt co-evolution method described above, the use of a validation-gated cache to manage LLM cost without compromising correctness, and the architectural separation between a stable public contract and a per-unit AI-generated implementation.

4.3 Limitations

The toolchain has several limitations that are worth stating explicitly. The RabbitMQ broker-backed system test exists in the test directory but is excluded from CI because it requires a live broker. CI exercises the send cmdlet only via mocks, and the integration test only covers the construct-validate handshake, not the final step of publishing the event to the broker. A broker-backed CI stage would strengthen the production-readiness claim and is the most prominent omission.

For two earlier versions of EiffelArtifactCreatedEvent, the schema uses version as a GAV identity field. At the moment, the top-level facade maps both the GAV version and the schema version to the same parameter. This works in the current tests, but it is semantically incorrect if a user wants to send an artifact whose GAV version differs from the schema version. This can be solved through renaming the affected parameter in the generator and then regenerating the two impacted modules.

A cross-version parameter naming inconsistency exists where the same field appears under different parameter names across schema versions because the Eiffel schemas themselves drifted. Tab completion mitigates the user impact, but a normalization layer that maps a canonical name to version-specific bindings has not been built.

The pre-flight validator checks introduced in Stage 3 are deliberately conservative. It rejects only the degenerate case of a schema that requires fields combined with a cmdlet that exposes none. A stricter check, requiring the parameter count to equal the required-leaf count modulo known collapse rules, would catch a wider class of generation failures, at the risk of false positives on legitimate compositions.

The generator does not currently save retry data, so although each attempt is printed to the terminal, there is no summary showing which validator messages caused the retries. As a result, it is difficult to identify which schemas the LLM repeatedly struggles with and which ones it generates correctly on the first try. Even a simple log of the number of attempts per (event, version) pair would make it easier to focus prompt improvements on the schemas that cause the most issues, instead of relying on isolated failures observed during development.

4.4 Reflection on Sustainability

The project contributes to sustainability primarily by reducing repeated manual work and making the tooling easier to maintain over time. Instead of requiring each Eiffel event type and schema version to be written and updated by hand, the toolchain generates the PowerShell module directly from the official Eiffel JSON Schemas. This makes the schemas the single source of truth and reduces the risk that the implementation drifts away from the protocol specification when upstream changes occur. For an organisation such as Nexer, this creates economic and process value by lowering long-term maintenance costs and providing a reusable foundation that can be shared across several teams rather than rebuilt separately for each project.

There is also an environmental aspect to the way the toolchain is designed. AI-assisted generation introduces an additional compute cost, since each generation attempt requires LLM inference. However, the validated-cache mechanism introduced in stage 3 helps limit this impact by storing modules that have already passed validation and reusing them when the inputs have not changed. As a result, unnecessary regeneration can be avoided, which reduces both CI runtime and the associated LLM footprint.

Finally, the approach has social and organisational sustainability benefits. By lowering the barrier to producing correct Eiffel tooling, the project makes traceability infrastructure more accessible to teams that may not have the time or resources to build such tooling manually. This can support better observability, accountability, and consistency across the software development lifecycle. At the same time, the project maintains a responsible boundary around AI usage, as the AI may repair the generator, but it is not allowed to modify the human-written tests. This ensures that correctness remains anchored in human-defined requirements rather than being delegated entirely to AI-generated output, which is important for trustworthy AI-assisted software development.

4.5 Future Work

Several extensions follow naturally from the work presented here. The most immediate is a listener and consumer counterpart to the current publisher-only toolchain, subscribing to RabbitMQ, validating inbound events against the same version-specific schemas, and dispatching them to handlers. This would close the loop that the project's premise opens and make the toolchain usable for the full event-driven CI/CD architecture rather than only its sender side, aligning with the originally intended bidirectional usage pattern of Eiffel,

where consumers react to broadcast events to drive downstream CI/CD activity [12]. A complementary direction is link-graph validation, which would enforce the semantic rules of the Eiffel protocol that JSON Schema cannot express, such as the requirement that a CONTEXT link reference an event of a specific kind or that a CAUSE chain not contain cycles.

The generator is currently tied to OpenAI and to the model used during this project. This could be improved by introducing an abstraction layer for LLM providers. That would make it possible to compare generation quality, cost, repair frequency, and reliability across different models, including newer OpenAI models and comparable models from other providers, such as Claude, Gemini, or DeepSeek. Since the project already uses a committed cache, this change could be made without having to pay to regenerate every existing module.

Another possible improvement is an automatic schema update workflow connected to the upstream Eiffel schemas repository. When new or changed schemas are detected, the workflow could open pull requests and let the validator act as the cache gate, deciding which modules need to be regenerated.

Finally, the toolchain could be demonstrated in a real CI/CD pipeline. A reference integration where Send-EiffelEvent calls are used in a production-style workflow would help close the gap between the generated artifact and its intended industrial use case.

4.6 Closing Remarks

The contribution of this thesis is not that AI can write PowerShell code, but that a schema-driven architecture, paired with a deliberate separation between AI-generated specifics and hand-written abstractions, and disciplined by human-written tests and validator gates, can consistently produce tooling intended for industrial CI/CD environments that scales from a single event to a full protocol catalogue without proportionally scaling the manual engineering effort required. The result is a working foundation for Eiffel event integration on Windows platforms, and a documented pattern for the responsible incorporation of AI into specification-driven software engineering.

Bibliography

[1] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," IEEE Software, vol. 32, no. 2, pp. 50–54, Mar./Apr. 2015, doi: 10.1109/MS.2015.27.

[2] D. Ståhl, K. Hallén, and J. Bosch, "Continuous Integration and Delivery Traceability in Industry: Needs and Practices," in Proc. 2016 42nd Euromicro Conf. Software Eng. and Advanced Applications (SEAA), Limassol, Cyprus, 2016, pp. 68–72, doi: 10.1109/SEAA.2016.12.

[3] Microsoft, "PowerShell Documentation," Microsoft Learn. [Online].
Available: https://learn.microsoft.com/en-us/powershell/ (accessed May 11, 2026).

[4] Broadcom Inc., "RabbitMQ - Messaging that just works." [Online].
Available: https://www.rabbitmq.com/ (accessed May 11, 2026).

[5] OASIS, "OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0," OASIS Standard, Oct. 2012. [Online]. Available: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html

[6] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374v2, Jul. 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[7] Y. Liu, C. Tantithamthavorn, Y. Liu, and L. Li, "On the Reliability and Explainability of Language Models for Program Generation," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 5, Article 126, Jun. 2024, doi: 10.1145/3641540.

[8] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents," Internet-Draft draft-bhutton-json-schema-01, Internet Engineering Task Force, Jun. 2022. [Online]. Available: https://json-schema.org/draft/2020-12/json-schema-core

[9] 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 Advances in Neural Information Processing Systems (NeurIPS), vol. 36, 2023.

[10] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley Professional, 2010.

[11] R. Suter, "NJsonSchema for .NET." [Online]. Available: https://github.com/RicoSuter/NJsonSchema (accessed May 11, 2026).

[12] D. Ståhl, K. Hallén, and J. Bosch, "Achieving Traceability in Large Scale Continuous Integration and Delivery Deployment, Usage and Validation of the Eiffel Framework," Empirical Software Engineering, vol. 22, no. 2, pp. 967–995, Apr. 2017, doi: 10.1007/s10664-016-9457-1.

Appendix

Appendix A: Markdown code for prompt builder

```
import json
import re

# These meta fields are always auto-generated and must never become user
parameters.
_AUTO_GENERATED = frozenset({'meta.id', 'meta.type', 'meta.version', 'meta.time'})


def _safe_version(version: str) -> str:
    return version.replace(".", "_")


def function_names(event_name: str, version: str) -> dict[str, str]:
    safe_ver = _safe_version(version)
    return {
        "new": f"New-{event_name}V{safe_ver}",
        "test": f"Test-{event_name}V{safe_ver}",
        "send": f"Send-{event_name}V{safe_ver}",
    }


def _collect_required_paths(schema: dict) -> list[str]:
    paths: list[str] = []

    def walk(node: dict, prefix: str) -> None:
        """
        Walk the schema node at `prefix`, collecting all required field paths.
        `prefix` is the dot-separated path to this node (may contain [] for array
items).
        """
        node_type = node.get("type")

        if node_type == "object":
            required = node.get("required")
            properties = node.get("properties", {})
            # Some Eiffel schemas (e.g. EiffelIssueDefinedEvent) omit the top-level
            # `required` array even though sub-objects like `data` declare their
            # own required fields. At the schema root only, descend into every
            # nested OBJECT property that has its own `required` array so the
            # inner required leaves still surface as cmdlet parameters. Arrays
            # whose top-level presence isn't required are skipped (they may be
            # legitimately empty).
            if required is None:
                if prefix == "":
                    required = [
                        k for k, p in properties.items()
                        if isinstance(p, dict)
                        and p.get("type") == "object"
                        and "required" in p
                    ]
                else:
                    required = []
            for key in required:
                child = properties.get(key)
```

```
                child_prefix = f"{prefix}.{key}" if prefix else key
                if not isinstance(child, dict):
                    paths.append(child_prefix)
                    continue
                walk(child, child_prefix)
            return

        if node_type == "array":
            items = node.get("items", {})
            array_prefix = prefix + "[]"
            if isinstance(items, dict):
                if items.get("type") == "object":
                    required = items.get("required", [])
                    properties = items.get("properties", {})
                    for key in required:
                        child = properties.get(key)
                        child_prefix = f"{array_prefix}.{key}"
                        if not isinstance(child, dict):
                            paths.append(child_prefix)
                            continue
                        # Recurse: handles nested objects and arrays inside array
items.
                        walk(child, child_prefix)
                else:
                    paths.append(array_prefix)
            else:
                paths.append(array_prefix)
            return

        # Primitive leaf — prefix already holds the full path.
        paths.append(prefix)

    walk(schema, "")
    return sorted(set(p for p in paths if p))


def _singularize(word: str) -> str:
    """
    Best-effort singularization for array field names found in JSON schemas.

    Handles the common regular plural forms seen in Eiffel schemas.
    Leaves words unchanged when the result would be ambiguous or wrong
    (e.g. status, process, class stay as-is).
    """
    if word.endswith('ies') and len(word) > 4:
        return word[:-3] + 'y'           # activities -> activity
    if word.endswith('s') and not word.endswith('ss') \
            and not word.endswith('us') and not word.endswith('is') \
            and not word.endswith('as') and len(word) > 3:
        return word[:-1]                 # locations -> location, links -> link,
tags -> tag
    return word                          # status, process, class, etc. unchanged


# Reserved names on Send-* and Test-* cmdlets. If a creator-mirror parameter
# would collide with one of these, _derive_param_name prefixes the leaf with
# its parent path segment to disambiguate (e.g. data.uri -> DataUri, not Uri,
# because -Uri is the reserved AMQP connection parameter on Send-*).
_RESERVED_NAMES = {
    'Payload', 'Uri', 'RabbitMQHost', 'RabbitMQPort', 'RabbitMQVirtualHost',
```

```
    'UserName', 'Password', 'ValidateBeforeSend',
    'Event', 'Version',
}


def _derive_param_name(path: str) -> str:
    """
    Derive a deterministic PascalCase PowerShell parameter name from a required
schema path.

    Rules:
    - If path contains at least one []:
        Take the singular form of the LAST array name, capitalize it, then append
all
        path segments after the last [] (each segment capitalized).
        data.locations[].type                           -> LocationType
        links[].target                                  -> LinkTarget
        data.fileInformation[].integrityProtection.alg ->
FileInformationIntegrityProtectionAlg
        data.batches[].recipes[].testCase.id            -> RecipeTestCaseId
    - Otherwise (nested object or scalar field):
        Use the leaf field name with the first letter uppercased.
        meta.security.authorIdentity -> AuthorIdentity
    - Collision guard: if the derived name clashes with a reserved Send-*/Test-*
      parameter name (Uri, RabbitMQHost, ...), prefix it with the parent path
      segment to disambiguate. data.uri -> DataUri.
    """
    if '[]' in path:
        # Split on every [] occurrence.
        # e.g. "data.batches[].recipes[].testCase.id" ->
        #   ["data.batches", ".recipes", ".testCase.id"]
        parts = re.split(r'\[\]', path)
        # The last array name is the final dot-segment of parts[-2].
        last_array_segment = parts[-2].split('.')[-1]
        singular = _singularize(last_array_segment)
        # Sub-fields following the last []: strip leading dot, split by dot.
        after = parts[-1]  # e.g. ".integrityProtection.alg" or ".type"
        sub_fields = [s for s in after.split('.') if s]
        all_parts = [singular] + sub_fields
        name = ''.join(s[0].upper() + s[1:] for s in all_parts)
    else:
        leaf = path.split('.')[-1]
        name = leaf[0].upper() + leaf[1:]

    if name in _RESERVED_NAMES:
        segments = path.split('.')
        if len(segments) >= 2:
            parent = segments[-2].replace('[]', '')
            if parent:
                name = parent[0].upper() + parent[1:] + name

    return name


def user_param_mapping(required_paths: list[str]) -> list[tuple[str, str]]:
    """
    Return (schema_path, parameter_name) pairs for all required paths that are
    not auto-generated. Names are computed deterministically by _derive_param_name.

    Raises ValueError if two distinct paths produce the same parameter name, since
```

```
    that would silently create a broken PowerShell module with duplicate
parameters.
    """
    seen: dict[str, str] = {}  # name -> first path that produced it
    result: list[tuple[str, str]] = []
    for path in required_paths:
        if path in _AUTO_GENERATED:
            continue
        name = _derive_param_name(path)
        if name in seen:
            raise ValueError(
                f"Duplicate PowerShell parameter name '{name}' derived from paths "
                f"'{seen[name]}' and '{path}'. "
                "Extend _derive_param_name to distinguish them."
            )
        seen[name] = path
        result.append((path, name))
    return result


def build_ps_module_prompt(event_name: str, version: str, schema_json: str) -> str:
    names = function_names(event_name, version)
    schema = json.loads(schema_json)
    required_paths = _collect_required_paths(schema)
    params = user_param_mapping(required_paths)

    if params:
        param_block = "\n".join(
            f"   - schema path: {path}  →  parameter name: {name}"
            for path, name in params
        )
    else:
        param_block = "   - (none — all required fields are auto-generated)"

    return f"""
You are an expert PowerShell engineer.
Generate a SINGLE Windows PowerShell module (.psm1) for Eiffel event {event_name}
version {version}.

Schema source of truth (must be respected):
{schema_json}

ABSOLUTE OUTPUT RULES:
- Output ONLY valid PowerShell module code.
- No markdown and no explanations.
- Export exactly these functions:
  - {names["new"]}
  - {names["test"]}
  - {names["send"]}
- Module must import successfully.

FUNCTION CONTRACT:
1) {names["new"]}
   - Builds and returns a PowerShell hashtable Eiffel payload.
   - Must include required fields for a valid {event_name} {version} payload.
   - Must auto-generate:
       meta.id      = GUID string
       meta.type    = "{event_name}"
       meta.version = "{version}"
       meta.time    = current UTC epoch milliseconds
```

```
   - Expose exactly these mandatory PowerShell parameters (one per required schema
path):
{param_block}
   - Use EXACTLY the parameter names listed above. Do not rename, abbreviate, or
merge them.
   - Map each parameter to the schema path shown next to it.
   - DO NOT expose optional fields as parameters.
   - DO NOT include optional fields in output payload.
   - For required arrays of objects:
       - create exactly one object item
       - expose parameters only for required item fields
   - For required enum fields:
       - ALWAYS expose a parameter with [ValidateSet(...)] containing all schema
enum values
       - never hardcode enum values in payload construction

2) {names["test"]}
   - Parameters:
       Payload (hashtable, optional)
   - If Payload is not provided, call {names["new"]}.
   - If {names["new"]} requires mandatory fields, expose the same mandatory input
     parameters on {names["test"]} and pass them through to {names["new"]}.
   - PARAMETER SETS (CRITICAL):
       - Declare EXACTLY two parameter sets so the cmdlet is callable both
         "from a payload" and "by constructing one from creator parameters":
           - "ByPayload" — DEFAULT — only -Payload (Mandatory in this set)
           - "Create"   — every creator-mirror mandatory parameter (Mandatory in
this set)
       - Mark the cmdlet with [CmdletBinding(DefaultParameterSetName='ByPayload')]
       - On -Payload: [Parameter(Mandatory=$true, ParameterSetName='ByPayload')]
       - On every creator-mirror parameter: [Parameter(Mandatory=$true,
ParameterSetName='Create')]
       - This ensures `{names["test"]} -Payload $p` binds unambiguously to
ByPayload
         and never reports "missing mandatory parameters".
   - Import module by resolving repo root from generated_modules path.
   - Use EXACT pattern:
       $repoRoot = Resolve-Path (Join-Path $PSScriptRoot "..\\..\\..")
       $validateModulePath = Join-Path $repoRoot "tool_generator\\ps\\Validate-
EiffelEventPayload.psm1"
       Import-Module $validateModulePath -Force
   - Call:
       Test-EiffelEventPayload -Payload <payload> -EventName "{event_name}" -
Version "{version}"
   - Return boolean result.
   - If Payload is provided, validate that Payload directly and do not pass creator
     parameters through (avoid empty-string ValidateSet binding errors).

3) {names["send"]}
   - Parameters:
       Payload (hashtable, optional)
       ValidateBeforeSend (switch)
       Uri (uri, optional)
       RabbitMQHost (string, default 127.0.0.1)
       RabbitMQPort (int, default 5672)
       RabbitMQVirtualHost (string, default /)
       UserName (string, optional)
       Password (string, optional)
   - If Payload is not provided, call {names["new"]}.
   - If {names["new"]} requires mandatory fields, expose the same mandatory input
```

```
     parameters on {names["send"]} and pass them through to {names["new"]}.
   - PARAMETER SETS (CRITICAL):
       - Declare EXACTLY two parameter sets so the cmdlet is callable both
         "from a payload" and "by constructing one from creator parameters":
           - "ByPayload" — DEFAULT — -Payload (Mandatory) plus all the optional
             AMQP connection params (-Uri, -RabbitMQHost, -RabbitMQPort,
             -RabbitMQVirtualHost, -UserName, -Password, -ValidateBeforeSend)
           - "Create"   — every creator-mirror mandatory parameter (Mandatory),
             plus the SAME optional AMQP connection params
       - Mark the cmdlet with [CmdletBinding(DefaultParameterSetName='ByPayload')]
       - On -Payload: [Parameter(Mandatory=$true, ParameterSetName='ByPayload')]
       - On every creator-mirror parameter: [Parameter(Mandatory=$true,
ParameterSetName='Create')]
       - On every optional AMQP connection parameter and -ValidateBeforeSend:
         declare TWO ParameterAttributes — one per set — both Mandatory=$false.
         Do NOT use __AllParameterSets implicitly by omitting ParameterSetName,
         because that confuses set resolution when only -Payload is supplied.
   - If ValidateBeforeSend is set, call {names["test"]}.
   - When calling {names["test"]} from {names["send"]}, pass only -Payload when
     Payload is already available.
   - Import module by resolving repo root from generated_modules path.
   - Use EXACT pattern:
       $repoRoot = Resolve-Path (Join-Path $PSScriptRoot "..\\..\\..")
       $sendModulePath = Join-Path $repoRoot "tool_generator\\ps\\Send-
EiffelEventToRabbitMq.psm1"
       Import-Module $sendModulePath -Force
   - Call Send-EiffelEventToRabbitMq with EXACT parameter names:
       -Event
       -Uri
       -RabbitMQHost
       -RabbitMQPort
       -RabbitMQVirtualHost
       -UserName
       -Password
   - NEVER use -Payload, -Host, -Port, or -VirtualHost when calling Send-
EiffelEventToRabbitMq.
   - SPLAT CONSTRUCTION: when forwarding parameters to Send-EiffelEventToRabbitMq,
     either pass each argument explicitly by name OR build ONE flat hashtable
     keyed by parameter name and splat it once. Do NOT rebuild the hashtable
     by piping through GetEnumerator/Where-Object/ForEach-Object — that pattern
     produces an array of single-key hashtables that pwsh splats incorrectly,
     yielding "A parameter cannot be found that matches parameter name
     'System.Collections.Hashtable+KeyCollection'". If you need to drop nulls,
     do it by removing keys from the original hashtable in place.
   - Return the send result.

IMPLEMENTATION RULES:
- Keep code deterministic and readable.
- Use hashtables and arrays that serialize cleanly to JSON.
- Do not include any external dependencies besides existing local PowerShell
modules.
- Ensure the generated module is compatible with existing project cmdlets exactly
by name/signature.
"""
```

Appendix B: Generated PowerShell module functionality

B.1: Runtime creation, validation, and sending of an EiffelArtifactPublishedEvent payload

```
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> import-Module .\EiffelEvents.psm1
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> $event = New-EiffelEvent EiffelArtifactPublishedEvent 4.0.1

cmdlet New-EiffelEvent at command pipeline position 1
Supply values for the following parameters:
LocationType: PLAIN
LocationUri: www.example.com
LinkTarget: 11111111-1111-4111-8111-111111111111
LinkType: ARTIFACT
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> test-EiffelEvent -Event EiffelArtifactPublishedEvent -Version
 4.0.1 -Payload $event
True
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> Send-EiffelEvent -Event EiffelArtifactPublishedEvent -Version
 4.0.1 -Payload $event -ValidateBeforeSend -Uri 'amqp://eiffel:eiffel@127.0.0.1:5672/'
True
```

B.2: Runtime creation, validation, and sending of an EiffelActivityFinishedEvent payload

```
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> import-Module .\EiffelEvents.psm1
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> $event = New-EiffelEvent EiffelActivityFinishedEvent 3.3.0

cmdlet New-EiffelEvent at command pipeline position 1
Supply values for the following parameters:
Conclusion: SUCCESSFUL
LinkTarget: 22222222-2222-4222-8222-222222222222
LinkType: ACTIVITY_EXECUTION
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> Test-EiffelEvent -Event EiffelActivityFinishedEvent -Version 3.3.
0 -Payload $event
True
PS C:\Users\Muhammad Usman\Desktop\eiffel_sender> Send-EiffelEvent -Event EiffelActivityFinishedEvent -Version 3.3.
0 -Payload $event -ValidateBeforeSend -Uri 'amqp://eiffel:eiffel@127.0.0.1:5672/'
True
```

B.3 RabbitMQ management view showing the published Eiffel event messages

Appendix C: EVENT_REGISTRY Configuration and Generated Module Structure

C.1: EVENT_REGISTRY configuration used by the generator

```
from pathlib import Path

DEFAULT_EVENT = "EiffelArtifactPublishedEvent"

REPO_ROOT = Path(__file__).resolve().parent.parent.parent
GENERATED_MODULE_ROOT = REPO_ROOT / "generated_modules"

# Event registry: event name -> version policy.
#   "all"    -> generate every schema version under schemas/<event>/
#   "latest" -> generate only the highest semver version
EVENT_REGISTRY: dict[str, str] = {
    "EiffelArtifactPublishedEvent": "all",
    "EiffelActivityCanceledEvent": "all",
    "EiffelActivityFinishedEvent": "all",
    "EiffelActivityStartedEvent": "all",
    "EiffelActivityTriggeredEvent": "all",
    "EiffelAnnouncementPublishedEvent": "all",
    "EiffelArtifactCreatedEvent": "all",
    "EiffelArtifactDeployedEvent": "all",
    "EiffelArtifactReusedEvent": "all",
    "EiffelCompositionDefinedEvent": "all",
    "EiffelConfidenceLevelModifiedEvent": "all",
    "EiffelEnvironmentDefinedEvent": "all",
    "EiffelFlowContextDefinedEvent": "all",
    "EiffelIssueDefinedEvent": "all",
    "EiffelIssueVerifiedEvent": "all",
    "EiffelTestCaseTriggeredEvent": "all",
    "EiffelTestCaseStartedEvent": "all",
    "EiffelTestCaseFinishedEvent": "all",
    "EiffelTestCaseCanceledEvent": "all",
    "EiffelTestSuiteStartedEvent": "all",
    "EiffelTestSuiteFinishedEvent": "all",
    "EiffelSourceChangeCreatedEvent": "all",
    "EiffelSourceChangeSubmittedEvent": "all",
    "EiffelTestExecutionRecipeCollectionCreatedEvent": "all",
}
```

C.2: Representative generated module structure for EiffelArtifactPublishedEvent

```
generated_modules/
  EiffelArtifactPublishedEvent/
    EiffelArtifactPublishedEvent.Facade.psm1
    v1_0_0/
      EiffelArtifactPublishedEvent.psm1
    v1_1_0/
      EiffelArtifactPublishedEvent.psm1
    v2_0_0/
      EiffelArtifactPublishedEvent.psm1
    v3_0_0/
      EiffelArtifactPublishedEvent.psm1
    v3_1_0/
      EiffelArtifactPublishedEvent.psm1
    v3_2_0/
      EiffelArtifactPublishedEvent.psm1
    v3_3_0/
      EiffelArtifactPublishedEvent.psm1
    v4_0_0/
      EiffelArtifactPublishedEvent.psm1
    v4_0_1/
      EiffelArtifactPublishedEvent.psm1

EiffelEvents.psm1
```

Appendix D: Schema-Alignment Test Suite

D.1: SchemaAlignment.Tests.ps1, independent schema-alignment tests for generated Eiffel modules

```
# Schema-alignment tests for the AI-generated per-version modules.
#
# These tests treat each event's JSON Schema as the ground truth and assert
# that the generated New-<Event>V<suffix> cmdlet faithfully reflects it. They
# exist because the existing matrix test is partially tautological: it
# introspects the cmdlet to decide which parameters to pass, then feeds the
# answer back to the cmdlet. That proves the module accepts what the module
# declares, not that the module declares what the schema requires.
#
# Three independent properties are checked per (event, version):
#
#   Coverage   - every required non-meta scalar leaf in the schema receives
#                a value in the emitted payload, originating from a mandatory
#                parameter (no required field is silently fabricated or
#                dropped).
#
#   RoundTrip  - every mandatory string parameter lands *somewhere* in the
#                emitted payload (no dead parameters that accept input and
#                discard it, no mis-wired parameters writing to nowhere).
#
#   Enum       - every multi-value enum on a required schema path has a
#                corresponding parameter whose ValidateSet matches the enum
#                as a set (no enum drift between schema and cmdlet surface).
#
# The schema walker here is intentionally reimplemented rather than imported
# from the generator: the point is to verify the generated module against the
# schema independently of the tool that produced it.

Set-StrictMode -Version Latest

Describe "Schema alignment of generated modules" {
    BeforeAll {
        $repoRoot                = (Resolve-Path (Join-Path $PSScriptRoot
"..\..")).Path
        $script:repoRoot         = $repoRoot
        $script:generatedRoot    = Join-Path $repoRoot "generated_modules"
        $script:schemasRoot      = Join-Path $repoRoot "schemas"

        # Walks a JSON-Schema object node, collecting:
        #   - leafPaths: dotted paths to every required scalar leaf reachable
        #                from the top-level required chain, using the
        #                convention "<name>[]" to denote "descend into array
        #                items" (i.e. the schema says the array must exist,
        #                and any item in it must satisfy items.required)
        #   - enums:    (path, enum list) pairs for multi-value enums on
        #                required paths
        function script:Walk-RequiredLeaves {
            param(
                $Node,
                [string]$Path,
                [System.Collections.ArrayList]$LeafPaths,
                [System.Collections.ArrayList]$Enums
            )
            if ($null -eq $Node) { return }
```

```
            if (-not ($Node -is [hashtable] -or $Node -is
[System.Collections.IDictionary])) { return }
            if (-not $Node.ContainsKey('type') -or $Node['type'] -ne 'object') {
return }
            if (-not $Node.ContainsKey('properties')) { return }

            $properties = $Node['properties']
            # Some Eiffel schemas (e.g. EiffelIssueDefinedEvent) omit the top-level
            # 'required' array even though sub-objects like `data` declare their
            # own required fields. At the schema root only, descend into every
            # nested OBJECT property that has its own 'required' array; arrays
            # whose top-level presence isn't required are deliberately skipped
            # (e.g. an empty `links: []` would be schema-valid, so we must not
            # assert links[].* exists). At inner levels a missing 'required'
            # genuinely means "no required leaves here".
            if (-not $Node.ContainsKey('required')) {
                if ($Path -ne '') { return }
                $synthetic = @()
                foreach ($k in $properties.Keys) {
                    $p = $properties[$k]
                    if (-not ($p -is [hashtable] -or $p -is
[System.Collections.IDictionary])) { continue }
                    if ($p['type'] -eq 'object' -and $p.ContainsKey('required')) {
$synthetic += $k }
                }
                $required = @($synthetic)
            }
            else {
                $required = @($Node['required'])
            }

            foreach ($name in $required) {
                if (-not $properties.ContainsKey($name)) { continue }
                $prop = $properties[$name]
                $sub  = if ($Path) { "$Path.$name" } else { $name }

                $type = $null
                if ($prop.ContainsKey('type')) { $type = $prop['type'] }

                if ($prop.ContainsKey('enum')) {
                    $enumValues = @($prop['enum'])
                    if ($enumValues.Count -gt 1) {
                        [void]$Enums.Add([pscustomobject]@{ Path = $sub; Enum =
$enumValues })
                    }
                }

                if ($type -eq 'object') {
                    script:Walk-RequiredLeaves -Node $prop -Path $sub -LeafPaths
$LeafPaths -Enums $Enums
                }
                elseif ($type -eq 'array' -and $prop.ContainsKey('items')) {
                    script:Walk-RequiredLeaves -Node $prop['items'] -Path "$sub[]"
-LeafPaths $LeafPaths -Enums $Enums
                }
                else {
                    # Scalar leaf (string/integer/number/boolean or a value with no
type).
                    [void]$LeafPaths.Add($sub)
                }
```

```
            }
        }

        # Resolves a dotted schema path against an emitted payload, returning
        # the flattened list of scalar values found at that path. Array
        # segments ("<name>[]") fan out across every array element.
        function script:Get-PayloadValuesAtPath {
            param($Payload, [string]$Path)
            $segments = $Path -split '\.'
            $cursor = @($Payload)
            foreach ($seg in $segments) {
                $next = New-Object System.Collections.ArrayList
                $isArraySeg = $false
                $key = $seg
                if ($seg.EndsWith('[]')) {
                    $isArraySeg = $true
                    $key = $seg.Substring(0, $seg.Length - 2)
                }
                foreach ($c in $cursor) {
                    if ($null -eq $c) { continue }
                    if (-not ($c -is [System.Collections.IDictionary])) { continue
}
                    if (-not $c.Contains($key)) { continue }
                    $value = $c[$key]
                    if ($isArraySeg) {
                        if ($value -is [System.Collections.IEnumerable] -and -not
($value -is [string])) {
                            foreach ($item in $value) { [void]$next.Add($item) }
                        }
                    }
                    else {
                        [void]$next.Add($value)
                    }
                }
                $cursor = $next.ToArray()
            }
            return $cursor
        }

        $script:commonParams = @(
            'Verbose','Debug','ErrorAction','WarningAction','InformationAction',

'ProgressAction','ErrorVariable','WarningVariable','InformationVariable',
            'OutVariable','OutBuffer','PipelineVariable','WhatIf','Confirm'
        )

        # Discover every (event, version) pair and prepare per-pair fixtures.
        $script:fixtures = @{}
        foreach ($eventDir in Get-ChildItem -Path $script:generatedRoot -Directory
-ErrorAction SilentlyContinue) {
            $eventName = $eventDir.Name
            foreach ($vDir in Get-ChildItem -Path $eventDir.FullName -Directory -
Filter 'v*' -ErrorAction SilentlyContinue) {
                $suffix  = $vDir.Name -replace '^v', ''
                $version = $suffix -replace '_', '.'

                $schemaPath = Join-Path $script:schemasRoot (Join-Path $eventName
"$version.json")
                if (-not (Test-Path $schemaPath)) { continue }
                $modulePath = Join-Path $vDir.FullName "$eventName.psm1"
```

```
                if (-not (Test-Path $modulePath)) { continue }

                Import-Module $modulePath -Force -DisableNameChecking

                $schema = Get-Content -Raw -Path $schemaPath | ConvertFrom-Json -
AsHashtable

                $leafPaths = New-Object System.Collections.ArrayList
                $enums     = New-Object System.Collections.ArrayList
                script:Walk-RequiredLeaves -Node $schema -Path '' -LeafPaths
$leafPaths -Enums $enums

                # Build call args with distinct sentinels per mandatory string
                # parameter so we can trace where each one lands in the payload.
                $newCmdName = "New-${eventName}V$suffix"
                $newCmd     = Get-Command $newCmdName -ErrorAction Stop

                $callArgs  = @{}
                $sentinels = @{}  # paramName -> sentinel string it was given
                $chosenEnums = @{}  # paramName -> chosen enum value

                foreach ($p in $newCmd.Parameters.Values) {
                    if ($script:commonParams -contains $p.Name) { continue }
                    $pa = $p.Attributes |
                        Where-Object { $_ -is
[System.Management.Automation.ParameterAttribute] } |
                        Select-Object -First 1
                    if (-not $pa -or -not $pa.Mandatory) { continue }

                    $vs = $p.Attributes |
                        Where-Object { $_ -is
[System.Management.Automation.ValidateSetAttribute] } |
                        Select-Object -First 1

                    if ($vs) {
                        $choice = $vs.ValidValues[0]
                        $callArgs[$p.Name] = $choice
                        $chosenEnums[$p.Name] = $choice
                    }
                    elseif ($p.ParameterType -eq [string]) {
                        # Sentinel shape: unique-per-param, string-searchable, and
                        # shaped to be inert for any pattern/format the schema
                        # might (loosely) enforce. We still pass valid-looking
                        # content for UUID/URI-shaped params so schema validation
                        # downstream does not choke.
                        $token = $null
                        if ($p.Name -match 'Target|Id$' -or $p.Name -match
'^.*Id$') {
                            # UUIDv4 with an embedded sentinel-friendly first
block.
                            $rand = [guid]::NewGuid().ToString()
                            $token = $rand
                        }
                        elseif ($p.Name -match 'Uri') {
                            $marker = [guid]::NewGuid().ToString("N")
                            $token = "https://example.test/rt/$marker"
                        }
                        else {
                            $token = "ROUNDTRIP_{0}_{1}" -f $p.Name,
[guid]::NewGuid().ToString("N")
```

```
                        }
                        $callArgs[$p.Name] = $token
                        $sentinels[$p.Name] = $token
                    }
                    elseif ($p.ParameterType -eq [int] -or $p.ParameterType -eq
[long] -or $p.ParameterType -eq [int64]) {
                        $num = Get-Random -Minimum 1000000000 -Maximum 2000000000
                        $callArgs[$p.Name] = $num
                        $sentinels[$p.Name] = "$num"
                    }
                    else {
                        $callArgs[$p.Name] = 'roundtrip-dummy'
                    }
                }

                $payload = & $newCmdName @callArgs

                $script:fixtures["$eventName@$version"] = [pscustomobject]@{
                    Event       = $eventName
                    Version     = $version
                    Suffix      = $suffix
                    Schema      = $schema
                    LeafPaths   = @($leafPaths)
                    Enums       = @($enums)
                    NewCmd      = $newCmd
                    CallArgs    = $callArgs
                    Sentinels   = $sentinels
                    ChosenEnums = $chosenEnums
                    Payload     = $payload
                }
            }
        }
    }

    Context "Coverage: every required non-meta schema leaf is populated" {
        BeforeDiscovery {
            $repoRoot = (Resolve-Path (Join-Path $PSScriptRoot "..\..")).Path
            $script:discoveredPairs = @()
            foreach ($eventDir in Get-ChildItem -Path (Join-Path $repoRoot
"generated_modules") -Directory -ErrorAction SilentlyContinue) {
                foreach ($vDir in Get-ChildItem -Path $eventDir.FullName -Directory
-Filter 'v*' -ErrorAction SilentlyContinue) {
                    $script:discoveredPairs += [pscustomobject]@{
                        Event   = $eventDir.Name
                        Version = ($vDir.Name -replace '^v', '' -replace '_', '.')
                    }
                }
            }
        }

        It "every required non-meta leaf in <Event> <Version> has a non-empty value
in the payload" -ForEach $script:discoveredPairs {
            $key = "$($_.Event)@$($_.Version)"
            $fx = $script:fixtures[$key]
            $fx | Should -Not -BeNullOrEmpty -Because "fixture for $key was built
in BeforeAll"

            $nonMetaLeaves = @($fx.LeafPaths | Where-Object { $_ -notmatch
'^meta(\.|$)' })
```

```
            $nonMetaLeaves.Count | Should -BeGreaterThan 0 -Because "every Eiffel
event has required non-meta fields"

            foreach ($leaf in $nonMetaLeaves) {
                $values = script:Get-PayloadValuesAtPath -Payload $fx.Payload -Path
$leaf
                $values.Count | Should -BeGreaterThan 0 -Because "required schema
leaf '$leaf' must exist in the payload for $key"
                foreach ($v in $values) {
                    ($null -ne $v -and "$v" -ne '') | Should -BeTrue -Because
"required schema leaf '$leaf' must carry a non-empty value for $key"
                }
            }
        }
    }

    Context "RoundTrip: every mandatory parameter value lands in the payload" {
        BeforeDiscovery {
            $repoRoot = (Resolve-Path (Join-Path $PSScriptRoot "..\..")).Path
            $script:discoveredPairs = @()
            foreach ($eventDir in Get-ChildItem -Path (Join-Path $repoRoot
"generated_modules") -Directory -ErrorAction SilentlyContinue) {
                foreach ($vDir in Get-ChildItem -Path $eventDir.FullName -Directory
-Filter 'v*' -ErrorAction SilentlyContinue) {
                    $script:discoveredPairs += [pscustomobject]@{
                        Event   = $eventDir.Name
                        Version = ($vDir.Name -replace '^v', '' -replace '_', '.')
                    }
                }
            }
        }

        It "every mandatory parameter sentinel for <Event> <Version> appears in the
emitted payload" -ForEach $script:discoveredPairs {
            $key = "$($_.Event)@$($_.Version)"
            $fx = $script:fixtures[$key]
            $fx | Should -Not -BeNullOrEmpty

            # Serialize once; string-search for each sentinel. If a sentinel is
            # missing, the parameter either does not flow into the payload at
            # all, or is being overwritten/ignored by the generator.
            $json = $fx.Payload | ConvertTo-Json -Depth 32 -Compress

            foreach ($paramName in $fx.Sentinels.Keys) {
                $token = $fx.Sentinels[$paramName]
                $json | Should -Match ([regex]::Escape($token)) -Because "mandatory
parameter '-$paramName' for $key must land somewhere in the payload"
            }
        }
    }

    Context "Enum: multi-value schema enums match a ValidateSet on the cmdlet" {
        BeforeDiscovery {
            $repoRoot = (Resolve-Path (Join-Path $PSScriptRoot "..\..")).Path
            $script:discoveredPairs = @()
            foreach ($eventDir in Get-ChildItem -Path (Join-Path $repoRoot
"generated_modules") -Directory -ErrorAction SilentlyContinue) {
                foreach ($vDir in Get-ChildItem -Path $eventDir.FullName -Directory
-Filter 'v*' -ErrorAction SilentlyContinue) {
                    $script:discoveredPairs += [pscustomobject]@{
```

```
                        Event   = $eventDir.Name
                        Version = ($vDir.Name -replace '^v', '' -replace '_', '.')
                    }
                }
            }
        }

        It "every multi-value required enum in <Event> <Version> has a matching
ValidateSet on the cmdlet" -ForEach $script:discoveredPairs {
            $key = "$($_.Event)@$($_.Version)"
            $fx = $script:fixtures[$key]
            $fx | Should -Not -BeNullOrEmpty

            if ($fx.Enums.Count -eq 0) {
                Set-ItResult -Skipped -Because "schema has no multi-value required
enums"
                return
            }

            # Collect every ValidateSet on the cmdlet up front (as sorted-comma
strings).
            $paramEnumSets = @()
            foreach ($p in $fx.NewCmd.Parameters.Values) {
                if ($script:commonParams -contains $p.Name) { continue }
                $vs = $p.Attributes |
                    Where-Object { $_ -is
[System.Management.Automation.ValidateSetAttribute] } |
                    Select-Object -First 1
                if (-not $vs) { continue }
                $joined = (($vs.ValidValues | Sort-Object) -join ',')
                $paramEnumSets += $joined
            }

            foreach ($e in $fx.Enums) {
                $schemaJoined = (($e.Enum | Sort-Object) -join ',')
                ($paramEnumSets -contains $schemaJoined) | Should -BeTrue `
                    -Because "schema enum at '$($e.Path)' for $key is
[$schemaJoined] but no cmdlet ValidateSet matches it as a set"
            }
        }
    }
}
```

Appendix E: CI Workflow and Reproducible Pipeline Definition

E.1: GitHub Actions CI workflow for generation, validation, packaging, and release

```
name: CI

on:
  push:
    branches:
      - main
      - powershell
    tags:
      - "v*"
    paths-ignore:
      - "generated_senders/**"
      - "generated_modules/**"
  pull_request:
    branches:
      - main
      - powershell
  workflow_dispatch:
    inputs:
      release_tag:
        description: "Optional release tag for manual release, for example v0.1.0"
        required: false
        type: string

permissions:
  contents: read

jobs:
  pytest:
    runs-on: ubuntu-latest
    env:
      OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"
          cache: pip
          cache-dependency-path: requirements.txt

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Generate missing senders
        if: ${{ env.OPENAI_API_KEY != '' }}
        run: python -m tool_generator.generate_missing

      - name: Run Python tests
        run: python -m pytest -q
```

```
  generate-module:
    runs-on: windows-latest
    needs: pytest
    env:
      OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Require OpenAI API key
        shell: pwsh
        run: |
          if ([string]::IsNullOrWhiteSpace($env:OPENAI_API_KEY)) {
            throw "OPENAI_API_KEY is required to generate the PowerShell module in
CI."
          }

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"
          cache: pip
          cache-dependency-path: requirements.txt

      - name: Install dependencies
        shell: pwsh
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Generate PowerShell modules (every event in EVENT_REGISTRY)
        shell: pwsh
        run: python -m tool_generator.ai_ps.run_loop_ps --all-events --max-attempts
3

      - name: Upload generated module artifact
        uses: actions/upload-artifact@v4
        with:
          name: generated-modules
          path: |
            generated_modules/
            EiffelEvents.psm1
          if-no-files-found: error
          retention-days: 7

  pester:
    runs-on: windows-latest
    needs: generate-module

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Download generated module artifact
        uses: actions/download-artifact@v4
        with:
          name: generated-modules
          path: .
```

```
      - name: Run PowerShell unit tests
        shell: pwsh
        run: |
          Invoke-Pester -Path tests/powershell/Validate-
EiffelEventPayload.Tests.ps1
          Invoke-Pester -Path tests/powershell/Send-EiffelEventToRabbitMq.Tests.ps1

      - name: Run install-layout contract test
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/Install-LayoutContract.Tests.ps1

      - name: Run packager gate test
        shell: pwsh
        run: Invoke-Pester -Path
          tests/powershell/Package-EiffelRuntimeArtifact.Tests.ps1

      - name: Run schema-alignment tests (coverage, round-trip, enum)
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/SchemaAlignment.Tests.ps1

      - name: Run tab-completion UX tests
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/TabCompletion.Tests.ps1

      - name: Run release-artifact end-to-end smoke
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/Release-ArtifactSmoke.Tests.ps1

      - name: Run PowerShell integration tests
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/Integration-EiffelFlow.Tests.ps1

      - name: Run top-level facade tests (event x version matrix)
        shell: pwsh
        run: Invoke-Pester -Path tests/powershell/EiffelEvents.Tests.ps1

      - name: Run per-event facade tests (once per event)
        shell: pwsh
        run: |
          $eventDirs = Get-ChildItem -Path "generated_modules" -Directory -
ErrorAction SilentlyContinue |
            Where-Object { Test-Path (Join-Path $_.FullName ("{0}.Facade.psm1" -f
$_.Name)) }
          if (-not $eventDirs) {
            throw "No event directories with a facade found under
generated_modules/"
          }
          $failed = @()
          foreach ($eDir in $eventDirs) {
            Write-Host "=== Running EventFacade.Tests.ps1 for $($eDir.Name) ==="
            $container = New-PesterContainer `
              -Path 'tests/powershell/EventFacade.Tests.ps1' `
              -Data @{ EventName = $eDir.Name }
            $result = Invoke-Pester -Container $container -PassThru
            if ($result.FailedCount -gt 0) {
              $failed += $eDir.Name
            }
          }
          if ($failed.Count -gt 0) {
            throw "Per-event facade tests failed for: $($failed -join ', ')"
```

```
          }

      - name: Run generated module behavior tests (every event x version)
        shell: pwsh
        run: |
          $eventDirs = Get-ChildItem -Path "generated_modules" -Directory -
ErrorAction SilentlyContinue |
            Where-Object { Test-Path (Join-Path $_.FullName ("{0}.Facade.psm1" -f
$_.Name)) }
          if (-not $eventDirs) {
            throw "No event directories with a facade found under
generated_modules/"
          }
          $failed = @()
          foreach ($eDir in $eventDirs) {
            $eventName = $eDir.Name
            $versionDirs = Get-ChildItem -Path $eDir.FullName -Directory -Filter
'v*' -ErrorAction SilentlyContinue
            foreach ($vDir in $versionDirs) {
              $version = $vDir.Name -replace '^v', '' -replace '_', '.'
              Write-Host "=== Running Pester for $eventName $version ==="
              $container = New-PesterContainer `
                -Path 'tests/powershell/Generated-EiffelEventModule.Tests.ps1' `
                -Data @{ EventName = $eventName; Version = $version }
              $result = Invoke-Pester -Container $container -PassThru
              if ($result.FailedCount -gt 0) {
                $failed += "$eventName@$version"
              }
            }
          }
          if ($failed.Count -gt 0) {
            throw "Pester tests failed for: $($failed -join ', ')"
          }

  release:
    runs-on: windows-latest
    needs: pester
    if: ${{ startsWith(github.ref, 'refs/tags/v') || (github.event_name ==
      'workflow_dispatch' && github.event.inputs.release_tag != '') }}
    permissions:
      contents: write

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Download generated module artifact
        uses: actions/download-artifact@v4
        with:
          name: generated-modules
          path: .

      - name: Resolve release tag
        id: release_tag
        shell: pwsh
        run: |
          if ("${{ github.event_name }}" -eq "workflow_dispatch") {
            $tag = "${{ github.event.inputs.release_tag }}"
          }
          else {
```

```
            $tag = "${{ github.ref_name }}"
          }

          if ([string]::IsNullOrWhiteSpace($tag)) {
            throw "Release tag is required."
          }

          "tag=$tag" >> $env:GITHUB_OUTPUT

      - name: Build Eiffel runtime artifact zip
        id: package
        shell: pwsh
        run: |
          New-Item -ItemType Directory -Path dist -Force | Out-Null
          .\scripts\Package-EiffelRuntimeArtifact.ps1 -OutputRoot (Resolve-Path
.\dist) -ReleaseTag "${{ steps.release_tag.outputs.tag }}"
          $zipPath = Resolve-Path ".\dist\eiffel-${{ steps.release_tag.outputs.tag
}}.zip"
          "zip_path=$zipPath" >> $env:GITHUB_OUTPUT

      - name: Publish GitHub release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        shell: pwsh
        run: |
          $tag = "${{ steps.release_tag.outputs.tag }}"
          $zipPath = "${{ steps.package.outputs.zip_path }}"

          gh release view $tag --repo "${{ github.repository }}" *> $null
          if ($LASTEXITCODE -eq 0) {
            gh release upload $tag $zipPath --clobber --repo "${{ github.repository
}}"
          }
          else {
            $notesLines = @(
              'Automated Eiffel message sender for power shell, built and verified
by CI.',
              '',
              '**Contents**',
              '- `EiffelEvents.psm1` — top-level multi-event facade exposing `New-
/Test-/Send-EiffelEvent` with `-Event` and `-Version` flags and tab completion on
both',
              '- Per-event facade (`<EventName>.Facade.psm1`) for single-event
pipelines; one per included event',
              '- Version-specific PowerShell modules for every included (event,
version) pair',
              '- **Full registry coverage:** all 24 supported Eiffel events with
every schema version (~224 (event, version) modules)',
              '    - Artifact: `EiffelArtifactPublishedEvent`,
`EiffelArtifactCreatedEvent`, `EiffelArtifactDeployedEvent`,
`EiffelArtifactReusedEvent`',
              '    - Activity: `EiffelActivityTriggeredEvent`,
`EiffelActivityStartedEvent`, `EiffelActivityFinishedEvent`,
`EiffelActivityCanceledEvent`',
              '    - Announcement / Composition:
`EiffelAnnouncementPublishedEvent`, `EiffelCompositionDefinedEvent`',
              '    - Confidence / Environment / Flow:
`EiffelConfidenceLevelModifiedEvent`, `EiffelEnvironmentDefinedEvent`,
`EiffelFlowContextDefinedEvent`',
              '    - Issue: `EiffelIssueDefinedEvent`, `EiffelIssueVerifiedEvent`',
```

```
              '    - Test case: `EiffelTestCaseTriggeredEvent`,
`EiffelTestCaseStartedEvent`, `EiffelTestCaseFinishedEvent`,
`EiffelTestCaseCanceledEvent`',
              '    - Test suite: `EiffelTestSuiteStartedEvent`,
`EiffelTestSuiteFinishedEvent`',
              '    - Source change: `EiffelSourceChangeCreatedEvent`,
`EiffelSourceChangeSubmittedEvent`',
              '    - Test execution:
`EiffelTestExecutionRecipeCollectionCreatedEvent`',
              '- Shared `Validate-EiffelEventPayload` and `Send-
EiffelEventToRabbitMq` modules',
              '- JSON Schemas for every included (event, version) pair',
              '- Runtime DLL dependencies (NJsonSchema, RabbitMQ.Client)',
              '- `Send-EiffelEventWithConfig.ps1` env-driven send wrapper',
              '- `Unblock-RuntimeFiles.ps1` helper for the Windows Mark-of-the-Web
step',
              '',
              '**Getting started**',
              'Extract the zip, open `pwsh` in the folder, run `Get-ChildItem -
Recurse -File | Unblock-File` once on Windows, then `Import-Module
.\EiffelEvents.psm1` and use `New-EiffelEvent -Event <Name> -Version <X>`. See
`README.md` for the full flag reference and the per-event alternative.',
              '',
              'No Python or OpenAI API key is required at runtime.'
            )
            $notes = $notesLines -join "`n"
            gh release create $tag $zipPath `
              --title "Eiffel $tag" `
              --notes $notes `
              --target "${{ github.sha }}" `
              --repo "${{ github.repository }}"
          }
```

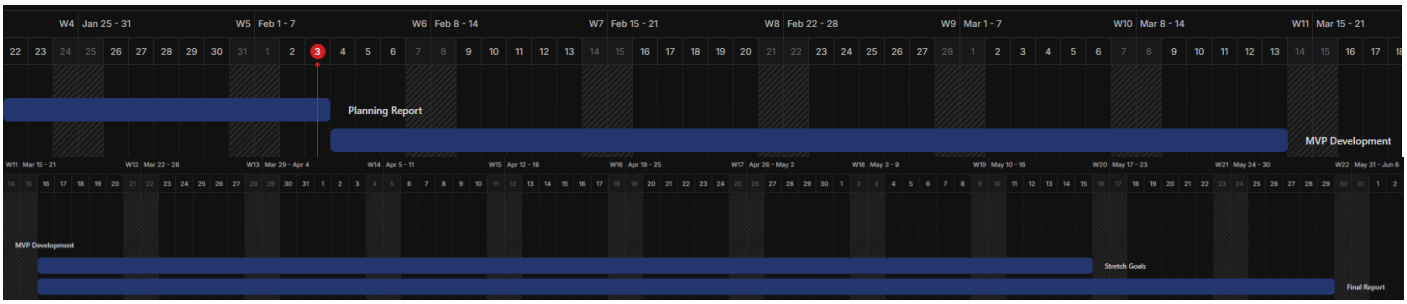
E.2: Scheduled schema-update workflow

```
name: Detect Schema Updates

on:
  schedule:
    - cron: "0 2 * * *"
  workflow_dispatch:

jobs:
  update-schemas:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout this repo
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Install dependencies
        run: pip install requests

      - name: Run schema fetcher
        run: python tool_generator/fetch_schemas.py

      - name: Check for schema changes
        id: changes
        run: |
          git config --global user.email "schema-bot@github.com"
          git config --global user.name "schema-bot"

          if git diff --quiet schemas/; then
            echo "changed=false" >> $GITHUB_ENV
          else
            echo "changed=true" >> $GITHUB_ENV
          fi

      - name: Stop if no changes
        if: env.changed == 'false'
        run: echo "No schema changes detected"

      - name: Create PR branch
        if: env.changed == 'true'
        run: |
          BRANCH="schema-update-$(date +'%Y%m%d%H%M%S')"
          echo "branch=$BRANCH" >> $GITHUB_ENV
          git checkout -b $BRANCH
          git add schemas/
          git commit -m "Schema update (auto)"
          git push origin $BRANCH

      - name: Create Pull Request
        if: env.changed == 'true'
        uses: peter-evans/create-pull-request@v5
        with:
          token: ${{ secrets.GITHUB_TOKEN }}
          branch: ${{ env.branch }}
```

```
          title: "Schema Update Detected"
          body: |
            This PR contains automatically fetched updates for Eiffel schemas.

            Once merged, this should trigger the sender generation pipeline.
          labels: auto-schema-update
```

Appendix F: Initial Project Timeline

F.1: Initial Gantt chart from the thesis planning report



Initial timeline as seen on the Gantt Chart above

- Planning report: 22/1 – 3/2
- MVP Development: 4/2 – 13/3
- Stretch Goals: 16/3 – 15/5
- Final Report (as well as Opposition report and Presentation): 16/3 – 29/5

The timeline above is an initial plan and subject to change over time based on the progress and or setbacks faced. The goal is to follow the Gantt Chart timeline but also remain flexible when required.

INSTITUTIONEN FÖR XXXX XXXXXX
XXXXXX XXXXXXXXXXX
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 20xx
www.chalmers.se