University of the Aegean

Department of Information and Communication Systems Engineering

Specification-Driven Testing (SDT): A Conceptual Framework for Automated API Quality Assurance in the GitOps Era

Diploma Thesis

Petros Evangelos Triantafyllis

Supervisor: Kyriakos Kritikos

Examination Committee:

Spyros Kokolakis, Professor, University of the Aegean

Ioannis E. Livieris, Assistant Professor, University of the Peloponnese

2026

Πανεπιστήμιο Αιγαίου

Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων

Έλεγχος Βάσει Προδιαγραφών (SDT):

Ένα Εννοιολογικό Πλαίσιο για την Αυτοματοποιημένη Διασφάλιση Ποιότητας API στην Εποχή του GitOps

Διπλωματική Εργασία

Πέτρος Ευάγγελος Τριανταφύλλης

Επιβλέπων: Κυριάκος Κρητικός

Εξεταστική Επιτροπή:

Σπύρος Κοκολάκης, Καθηγητής, Πανεπιστήμιο Αιγαίου

Ιωάννης Ε. Λιβιέρης, Επίκουρος Καθηγητής, Πανεπιστήμιο Πελοποννήσου

2026

Acknowledgements

I owe this work to three people, and it would not exist without each of them.

To **George Fiotakis**, my lead and mentor, who brought me into this domain and into the work that became this thesis. The direction this research took, and much of what I have learned about building software seriously, I owe to his guidance.

To **George Domalis**, in memoriam. He took me into his company, believed in me, and gave me the resources to do this work; he wanted the thesis to grow out of real engineering and to put that work forward, rather than to sit beside it. Beyond the opportunity, he gave me a way of thinking and an instinct for scale that shaped how I approach problems to this day. He was lost to an avalanche. This work carries his influence, and I dedicate it to his memory.

To my supervisor, **Kyriakos Kritikos**, who believed in me and whose every contribution was given with dedication and attention to detail, always in the service of bringing the work forward.

Abstract

The contemporary practice of API quality assurance is fragmented across loosely connected concerns: structural validation linters check whether an OpenAPI document parses, contract-testing tools verify that the API server matches its specification at a few hand-chosen endpoints, and load-testing harnesses measure performance under hand-authored scenarios. Each tool is independently useful; none addresses the underlying problem, which is that the specification itself is the only artefact common to development, testing, deployment, and operations, and that its quality determines the quality of every downstream automation that depends on it. This thesis proposes a different approach.

The proposal is developed as a three-layer contribution package: **SDT**, a conceptual framework that re-grounds API quality assurance in the specification itself; **DriveBy**, the reference implementation that makes the framework executable; and **XSDLC**, the GitOps integration that embeds it into the delivery pipeline as a declaratively provisioned quality gate. Each layer is summarised in turn below, followed by the empirical evaluation.

Specification-Driven Testing (SDT) is a conceptual framework for automated API quality assurance in which the specification is treated as the canonical source of truth for both *validation* (does the contract describe a well-formed, complete, secure API?) and *testing* (does the implementation satisfy the contract under realistic functional and load conditions?). The framework is anchored on three axioms—*Completeness*, *Determinism*, and *Observability*—and decomposed into nine principles (P001 through P009) covering structural compliance, documentation quality, error-handling, schema precision, security, functional correctness, performance, versioning, and test-readiness. The contribution at this layer is theoretical: a framework, not a complete methodology; the missing methodology elements (formal test-derivation procedure, execution and failure semantics, lifecycle governance) are scoped as future work.

DriveBy is the reference implementation of the framework: a Go command-line validator that accepts OpenAPI 3.0.x and 3.1.0 specifications

natively—and Swagger 2.0 documents through a transparent conversion adapter—runs the seven currently implemented principle checkers (P001–P005, P008, P009; P006 and P007 are scoped to the runtime test layer), and emits a deterministic, machine-readable JSON report with severity-classified findings. The CLI is built around an `APISpec` adapter that abstracts away version-specific details, allowing every validator to operate on a single semantic surface regardless of whether the underlying spec is OpenAPI 3.0.x, 3.1.0, or a converted Swagger 2.0 document.

XSDLC (the Crossplane Software Development Lifecycle composition) is the GitOps integration: a single Kubernetes custom resource that declaratively provisions a complete two-repository GitOps delivery pipeline—application repository, dedicated GitOps repository, ArgoCD applications with source-hydration overlays, the GitOps Promoter, multi-stage quality gates running DriveBy at staging and production boundaries, branch protection, and the ingress and certificate scaffolding required to run the system. XSDLC is the empirical artefact that demonstrates SDT can be embedded as a built-in feature of an Internal Developer Platform rather than a bolt-on linter.

The framework, the CLI, and the composition are evaluated empirically across four arms. A *controlled evaluation* on a hand-crafted reference API (`non-critical-api`) verifies defect detection at the level of individual checks: 12 of 15 injected single-check mutations are detected directly (an 80% detection rate; of the remainder, one mutation is neutralised by the loader's structural validation, one proves to be an injection artefact with no real defect, and exactly one is a genuine framework miss, documented as a limitation), and all 3 evaluable random multi-principle combinations are fully detected, every targeted principle flagged simultaneously. A *large-scale evaluation* on 50 OpenAPI 3.x specifications harvested from the APIs.guru registry shows that 84% of public APIs score 1/6 or worse in DriveBy strict mode—no API in the sample passes more than 2 of the 6 evaluated principles. An *operational proof-of-concept* deploys XSDLC on Novelcore's internal Kubernetes cluster and processes five APIs through the full delivery pipeline. Finally, an *agent-feedback experiment* on the canonical Swagger Petstore specification demonstrates that an AI coding agent, given only the DriveBy report as feedback, can autonomously revise a 1/6-scoring specification to 4/6 in a single iteration—empirical evidence for the framework's positioning as feedback infrastructure for agent-driven specification improvement.

The thesis closes with an explicit treatment of authorship and the development methodology, separating the author's intellectual ownership of the framework, architecture, and evaluation design from the AI-assisted productisation phase, with quantitative evidence drawn from the version-control history.

Keywords: API quality assurance, specification-driven testing, OpenAPI, Crossplane, GitOps, Kubernetes, quality gates, agent-driven development.

Περίληψη

Η σύγχρονη πρακτική της διασφάλισης ποιότητας των διεπαφών προγραμματισμού εφαρμογών (API) χαρακτηρίζεται από τον κατακερματισμό της σε διακριτά και χαλαρά συνδεδεμένα πεδία: τα εργαλεία στατικής ανάλυσης της δομής (structural validation linters) ελέγχουν απλώς αν ένα έγγραφο OpenAPI είναι συντακτικά ορθό, τα εργαλεία δοκιμών συμβολαίου (contract-testing tools) επαληθεύουν ότι ένα API συμμορφώνεται με τις προδιαγραφές του σε έναν περιορισμένο αριθμό επιλεγμένων τελικών σημείων (endpoints), ενώ τα συστήματα δοκιμών φόρτου (load-testing harnesses) αξιολογούν την απόδοση της εφαρμογής βάσει χειροκίνητα σχεδιασμένων σεναρίων. Αν και κάθε εργαλείο είναι χρήσιμο από μόνο του, κανένα δεν αντιμετωπίζει το ουσιαστικό πρόβλημα: η ίδια η προδιαγραφή (specification) αποτελεί το μοναδικό κοινό τεχνούργημα (artefact) μεταξύ της ανάπτυξης, των δοκιμών, της διάθεσης και της λειτουργίας, γεγονός που σημαίνει ότι η ποιότητά της καθορίζει άμεσα την ποιότητα κάθε επακόλουθης αυτοματοποίησης που βασίζεται σε αυτήν. Η παρούσα διατριβή προτείνει μια διαφορετική προσέγγιση στο ζήτημα αυτό.

Η πρόταση αναπτύσσεται ως ένα πακέτο συνεισφορών τριών επιπέδων: το **SDT**, ένα εννοιολογικό πλαίσιο που επαναθεμελιώνει τη διασφάλιση ποιότητας των API στην ίδια την προδιαγραφή· το **DriveBy**, η υλοποίηση αναφοράς που καθιστά το πλαίσιο εκτελέσιμο· και το **XSDLC**, η ενσωμάτωση GitOps που το εντάσσει στη ροή παράδοσης λογισμικού ως δηλωτικά παρεχόμενη πύλη ποιότητας. Τα τρία επίπεδα παρουσιάζονται συνοπτικά στη συνέχεια, ακολουθούμενα από την εμπειρική αξιολόγηση.

Ο **Έλεγχος Βάσει Προδιαγραφών** (Specification-Driven Testing, SDT) αποτελεί ένα εννοιολογικό πλαίσιο για την αυτοματοποιημένη διασφάλιση ποιότητας των API, στο πλαίσιο του οποίου η προδιαγραφή αντιμετωπίζεται ως η απόλυτη πηγή αλήθειας (canonical source of truth), τόσο για την *επικύρωση* (κατά πόσον το συμβόλαιο περιγράφει ένα καλοσχηματισμένο, πλήρες και ασφαλές API) όσο και για τον *έλεγχο* (κατά πόσον η υλοποίηση ικανοποιεί το συμβόλαιο υπό ρεαλιστικές συνθήκες λειτουργίας και φόρτου). Το πλαίσιο θεμελιώνεται σε τρία αξιώματα—*Πληρότητα*, *Ντετερμινισμός* και *Παρατηρησιμότητα*—και αναλύεται σε εννέα αρχές (P001 έως P009), οι οποίες

καλύπτουν τη δομική συμμόρφωση, την ποιότητα της τεκμηρίωσης, τον χειρισμό σφαλμάτων, την ακρίβεια των σχημάτων (schema precision), την ασφάλεια, τη λειτουργική ορθότητα, την απόδοση, τη διαχείριση εκδόσεων (versioning) και την ετοιμότητα για έλεγχο (test-readiness). Η συνεισφορά της διατριβής σε αυτό το επίπεδο είναι θεωρητικής φύσης, καθώς προτείνει ένα πλαίσιο και όχι μια ολοκληρωμένη μεθοδολογία· τα στοιχεία της μεθοδολογίας που παραλείπονται (τυπική διαδικασία εξαγωγής δοκιμών, σημασιολογία εκτέλεσης και αποτυχίας, διακυβέρνηση του κύκλου ζωής) ανάγονται σε μελλοντική έρευνα.

Το σύστημα DriveBy αποτελεί την υλοποίηση αναφοράς (reference implementation) του πλαισίου. Πρόκειται για ένα εργαλείο επικύρωσης γραμμής εντολών (command-line validator) αναπτυγμένο σε γλώσσα Go, το οποίο δέχεται εγγενώς προδιαγραφές OpenAPI 3.0.x και 3.1.0—καθώς και έγγραφα Swagger 2.0 μέσω διαφανούς προσαρμογέα μετατροπής—εκτελεί τους επτά μέχρι στιγμής υλοποιημένους ελεγκτές αρχών (P001–P005, P008, P009, καθώς οι P006 και P007 περιορίζονται στο επίπεδο ελέγχου χρόνου εκτέλεσης) και παράγει μια ντετερμινιστική, μηχανικά αναγνώσιμη αναφορά σε μορφή JSON, στην οποία τα ευρήματα ταξινομούνται βάσει της σοβαρότητάς τους. Η διεπαφή γραμμής εντολών (CLI) είναι δομημένη γύρω από έναν προσαρμογέα APISpec (APISpec adapter), ο οποίος παρέχει ένα επίπεδο αφαίρεσης (abstraction) για τις λεπτομέρειες που αφορούν συγκεκριμένες εκδόσεις του OpenAPI, επιτρέποντας σε κάθε ελεγκτή να λειτουργεί σε μια ενιαία σημασιολογική επιφάνεια, ανεξαρτήτως του αν η υποκείμενη προδιαγραφή αποτελεί έγγραφο OpenAPI 3.0.x, 3.1.0 ή αρχείο Swagger 2.0 που έχει προέλθει από μετατροπή.

Η σύνθεση XSDLC (Crossplane Software Development Lifecycle composition) αποτελεί την υλοποίηση ενσωμάτωσης της προσέγγισης GitOps: πρόκειται για έναν ενιαίο προσαρμοσμένο πόρο (custom resource) του Kubernetes, ο οποίος παρέχει με δηλωτικό τρόπο (declaratively provisions) μια πλήρη ροή παράδοσης λογισμικού (delivery pipeline) μέσω GitOps, βασισμένη σε αρχιτεκτονική δύο αποθετηρίων (repositories)—ένα αποθετήριο για την εφαρμογή, ένα αποκλειστικό αποθετήριο GitOps, εφαρμογές ArgoCD με επικαλύψεις ενυδάτωσης πηγαίου κώδικα (source-hydration overlays), τον GitOps Promoter, πύλες ποιότητας (quality gates) πολλαπλών σταδίων που εκτελούν το DriveBy κατά τη μετάβαση στα περιβάλλοντα δοκιμών (staging) και παραγωγής (production), μηχανισμούς προστασίας κλάδων (branch protection), καθώς και την απαραίτητη υποδομή δρομολόγησης (ingress) και διαχείρισης πιστοποιητικών για τη λειτουργία του συστήματος. Το XSDLC αποτελεί το εμπειρικό τεχνούργημα που αποδεικνύει ότι ο έλεγχος SDT μπορεί να ενσωματωθεί ως εγγενές χαρακτηριστικό μιας Εσωτερικής Πλατφόρμας Προγραμματιστών (Internal Developer Platform), αντί να αποτελεί ένα απλό πρόσθετο εργαλείο στατικής ανάλυσης (bolt-on linter).

Το πλαίσιο, η διεπαφή γραμμής εντολών (CLI) και η σύνθεση XSDLC α-

ξιολογούνται εμπειρικά σε τέσσερις άξονες. Πρώτον, μέσω μιας *ελεγχόμενης αξιολόγησης* σε ένα ειδικά σχεδιασμένο API αναφοράς (non-critical-api), η οποία επαληθεύει την ικανότητα ανίχνευσης μεμονωμένων ελαττωμάτων (ανιχνεύθηκαν άμεσα 12 από τις 15 μεταλλάξεις σε επίπεδο ελέγχου—ποσοστό ανίχνευσης 80%· από τις υπόλοιπες, μία μετάλλαξη εξουδετερώθηκε από τον δομικό έλεγχο του φορτωτή, μία αποδείχθηκε τεχνούργημα της έγχυσης χωρίς πραγματικό ελάττωμα, και ακριβώς μία συνιστά γνήσια αστοχία του πλαισίου, τεκμηριωμένη ως περιορισμός· επιπλέον, εντοπίστηκαν πλήρως και οι 3 αξιολογήσιμοι τυχαίοι συνδυασμοί προβλημάτων που αφορούν παραβίαση πολλαπλών αρχών). Δεύτερον, πραγματοποιήθηκε μια *αξιολόγηση μεγάλης κλίμακας* σε 50 προδιαγραφές OpenAPI 3.x, οι οποίες αντλήθηκαν από το μητρώο APIs.guru. Τα αποτελέσματα καταδεικνύουν ότι το 84% των δημόσιων API λαμβάνει βαθμολογία 1/6 ή χαμηλότερη στην αυστηρή λειτουργία (strict mode) του DriveBy, ενώ κανένα API του δείγματος δεν ικανοποιεί περισσότερες από 2 εκ των 6 αξιολογούμενων αρχών. Τρίτον, μέσω μιας *λειτουργικής απόδειξης έννοιας* (operational proof-of-concept), το XSDLC αναπτύχθηκε στο εσωτερικό cluster Kubernetes της Novelcore, επεξεργαζόμενο επιτυχώς πέντε διαφορετικά API μέσα από την πλήρη ροή παράδοσης. Τέταρτον, διεξήχθη ένα *πείραμα ανατροφοδότησης πρακτόρων* (agent-feedback experiment) χρησιμοποιώντας την καθιερωμένη προδιαγραφή Swagger Petstore. Το πείραμα αυτό αποδεικνύει ότι ένας πράκτορας προγραμματισμού τεχνητής νοημοσύνης (AI coding agent), λαμβάνοντας ως μοναδική ανατροφοδότηση την αναφορά του DriveBy, δύναται να αναθεωρήσει αυτόνομα μια προδιαγραφή, βελτιώνοντας τη βαθμολογία της από 1/6 σε 4/6 σε μία μόλις επανάληψη. Το γεγονός αυτό παρέχει εμπειρικά τεκμήρια που υποστηρίζουν τη χρήση του πλαισίου ως υποδομή ανατροφοδότησης για τη βελτίωση προδιαγραφών που καθοδηγείται από πράκτορες (agent-driven specification improvement).

Η εργασία ολοκληρώνεται με εκτενή αναφορά στο ζήτημα της πνευματικής ιδιοκτησίας και στη μεθοδολογία ανάπτυξης. Συγκεκριμένα, διαχωρίζεται η πνευματική ιδιοκτησία (intellectual ownership) του συγγραφέα—όσον αφορά το πλαίσιο, την αρχιτεκτονική και τον σχεδιασμό της αξιολόγησης—από τη φάση ανάπτυξης και προτυποποίησης (productisation) που πραγματοποιήθηκε με την υποστήριξη τεχνητής νοημοσύνης, παραθέτοντας σχετικά ποσοτικά τεκμήρια τα οποία αντλούνται από το ιστορικό ελέγχου εκδόσεων (version-control history).

Λέξεις-κλειδιά: διασφάλιση ποιότητας API, έλεγχος βάσει προδιαγραφών, OpenAPI, Crossplane, GitOps, Kubernetes, πύλες ποιότητας, ανάπτυξη καθοδηγούμενη από πράκτορες.

Contents

1 Introduction **20**
1.1 The Cost of Manual Operations 21
1.2 The Shift Toward Declarative Everything 22
1.3 Structured Knowledge vs. Tribal Knowledge 23
1.4 Problem Statement 25
1.5 Research Questions 25
1.6 Thesis Contributions 27
1.7 A Technology Showcase 29
1.8 Thesis Structure 29

2 Related Work **31**
2.1 The OpenAPI Ecosystem 31
2.2 API Testing: Foundations and Taxonomy 33
2.3 GitOps and Continuous Delivery 36
2.4 Infrastructure as Code and the Declarative Paradigm 38
2.5 AI-Assisted Software Engineering 42
2.6 Synthesis and Research Gap 43

3 A Conceptual Framework: Specification-Driven Testing **48**
3.1 The SDT Paradigm 48
3.2 The Three Axioms 49
3.3 The Nine Principles 52
3.4 Severity, Status, and the Gate Decision 62
3.5 Validation Modes 63
3.6 Summary 65

4 System Architecture **66**
4.1 System Context 66
4.2 The Three-Layer Model 67
4.3 From CLI Tool to Custom Resource 68
4.4 Chapter Roadmap 70

5 The DriveBy CLI: Internal Architecture **71**
5.1 Architecture Overview 71
5.2 The Type System 75
5.3 The Specification Abstraction Layer 76
5.4 Specification Loading and Preprocessing 81
5.5 The Principle Checker Pattern 83
5.6 The Validation Engine 88
5.7 Runtime Testing Subsystem 88
5.8 Report Generation 91
5.9 Command-Line Interface 92
5.10 Summary 94

6 Kubernetes Architecture **95**
6.1 Architecture Overview 95
6.2 The Kubernetes Controller Pattern 96
6.3 Crossplane XRD Design 99
6.4 Declarative Lifecycle Management 106
6.5 Composition Pipeline 109
6.6 Two-Tier Configuration Model 112
6.7 Argo Events Architecture 113
6.8 Argo Workflows Validation DAG 115
6.9 Helm Chart Packaging 119
6.10 Summary 121

7 GitOps Pipeline and CI/CD **122**
7.1 Promotion-Driven Quality Gate 122
7.2 Event-Driven Validation Pipeline 123
7.3 Two-Repo Architecture 124
7.4 Multi-Environment Promotion 128
7.5 Human-in-the-Loop Patterns 130
7.6 Quality Gate as Control Loop 134
7.7 CI/CD of DriveBy Itself 139
7.8 Operational Deployment 141
7.9 Summary 143

8 Evaluation **144**
8.1 Evaluation Framework 144
8.2 Operational Evaluation: Proof-of-Concept Deployment 146
8.3 Controlled Evaluation: non-critical-api 154
8.4 Multi-API Controlled Evaluation 158
8.5 Large-Scale Evaluation: APIs.guru 164

8.6 SDT Feedback as Input to Agent-Driven Specification Improvement . . . 169
8.7 Summary . . . 173

9 AI-Assisted Development: Authorship, Process, and Reflection 174
9.1 Authorship and the Author's Role . . . 174
9.2 The CLAUDE.md Knowledge Protocol . . . 180
9.3 Persistent Agent Memory . . . 183
9.4 Agent-Driven Multi-API Evaluation . . . 184
9.5 Agent-Driven Thesis Writing . . . 186
9.6 SDT as Agent Feedback Infrastructure . . . 187
9.7 Personal Lessons Learned . . . 189
9.8 Summary . . . 190

10 Discussion 191
10.1 Revisiting the Research Questions . . . 191
10.2 The Specification Quality Gap . . . 195
10.3 The BYOCI Boundary . . . 197
10.4 Threats to Validity . . . 197
10.5 Implications for Practice . . . 199
10.6 Summary . . . 200

11 Conclusion 201
11.1 Summary of Contributions . . . 201
11.2 Answers to Research Questions . . . 202
11.3 XSDLC as Ontological Specification . . . 204
11.4 Limitations . . . 205
11.5 Future Work . . . 207

List of Figures

4.1 System context. The dashed boundary encloses the system under study: the DriveBy CLI (validation engine), the XSDLC composition (pipeline generator), the generated gate infrastructure, and the GitOps Promoter (promotion controller), resting on the infrastructure substrate that hosts and reconciles it. External entities: human actors, the target API and its specification, the GitHub repositories, and the validation report. 67

5.1 DriveBy package dependency graph. Arrows indicate import direction; colour denotes architectural layer. 73

5.2 Component diagram of the DriveBy CLI. Solid arrows are internal package dependencies; dashed violet arrows cross the system boundary to external interfaces. Colour denotes architectural layer (data, validation, interface), matching Figure 5.1; the off-chain `report` package is assigned to the interface layer. The cross-cutting packages `util`, `github`, and `logger` (Table 5.1) are omitted. 74

5.3 Class / interface diagram of DriveBy's two central abstractions. Dashed grey arrows denote interface realisation (concrete class implements interface); solid grey arrows denote use. Dashed nodes (P006, P007) are realised by the runtime testing subsystem (Section 5.7); their `PrincipleChecker` wrappers are pending. 84

6.1 Kubernetes deployment view of the target cluster (thesis-built vs. inherited components). 97

6.2 Dual-provider architecture. The XSDLC composition delegates in-cluster resources to `provider-kubernetes` (blue) and GitHub resources to `provider-upjet-github` (orange); the composition layer itself (violet) is built in this thesis. Colours encode the target control plane that owns each resource; see the legend at the bottom of the figure. 100
6.3 Resources generated by the `XSDLC` XRD. App-level shared resources (ServiceAccount, EventBus, ScmProvider, GitRepository, Repository, RepositoryFiles, push secret) are created once per CR. Per-environment resources (PromotionStrategy, Branches, overlay RepositoryFile, ghcr-creds, ArgoCD Application) are created for each environment. Per-gate resources (Role, RoleBinding, WorkflowTemplate, EventSource, Sensor, Service, Ingress, BranchProtection) are created for each environment that defines a quality gate. 105
6.4 Crossplane reconciliation loop. The XSDLC spec feeds the composition, which renders desired children. Providers reconcile each child against its target system. Status feedback flows back through `function-auto-ready` to mark the `XSDLC` composite resource as ready. 107
6.5 Declarative lifecycle management. Day 0 installs the platform; Day 1 creates the pipeline from a single CR; Day 2 and beyond provide continuous reconciliation with drift detection and self-healing; Day N supports updates and clean deletion. 109
6.6 Two-tier configuration resolution. Tier 2 (XRD spec) overrides values from Tier 1 (Helm defaults), with the XRD spec having the highest priority. 112

7.1 End-to-end GitOps validation pipeline. The developer commits dry manifests to the `main` branch of the gitops repository (BYOCI boundary). The ArgoCD Source Hydrator renders per-environment overlays and writes hydrated output to `-next` branches. The Promoter creates a promotion PR; the webhook triggers Argo Events, which submits an Argo Workflow. Validation results are posted back as commit status and PR comment (dashed feedback loop). 123

7.2 Two-repo branch topology. The developer commits dry manifests to `main` in the auto-created GitOps repository. The ArgoCD Source Hydrator renders per-environment overlays and writes hydrated output to `-next` branches. The Promoter creates PRs to active branches; quality gates control promotion (gate names are indicative—each environment configures its own checks); ArgoCD auto-syncs from active branches. The fan-out to all three `-next` branches depicts a shared-base change; a single-overlay commit re-hydrates only its environment. The software repository is untouched—the BYOCI boundary separates developer CI from XSDLC delivery entirely. 127
7.3 Multi-environment promotion model. Development auto-syncs; staging promotion is gated by SDT validation (commit status); production requires manual approval. 128
7.4 Modular automation spectrum. Each environment is a pipeline of four stages with two independently configurable slots: quality gate (`gate.checks[]`) and merge policy (`autoMerge`). ArgoCD always auto-syncs. The composition renders different infrastructure for each combination—from fully automated (Env_1: no gate, auto-merge, auto-sync) through gated automation (Env_2: SDT + functional test, auto-merge) to human-controlled merge (Env_N: SDT + load test, manual approval, auto-sync). The Propagate and Sync stages are fixed; quality gate and merge policy are configurable per environment. . . . 132
7.5 Quality gate as a classical control loop. The promotion-policy predicate (OpenAPI spec + XSDLC manifest) is the setpoint; the delivery system is the plant; the DriveBy CLI is the sensor, its validation verdict the measurement; the Argo Workflow is the controller; CommitStatus and Promoter form the actuator that merges or blocks the promotion PR. Negative feedback blocks non-conforming transitions. The platform tier is collapsed into the single "Controller" block; Section 7.6.2 and Figure 7.6 decompose it into the stack of CRDs and reconcilers that realise it. 135

7.6 The controller stack. The verdict tier (DriveBy) is open-loop: it emits a structured report but performs no repair on the API or the spec. The platform tier is closed-loop and autonomous: Crossplane composes the managed objects from the XSDLC manifest and orchestrates a stack of controllers, each performing continuous reconciliation within its CRD domain. The captions under each platform-tier box name the dominant repair action each controller performs when its declared state diverges from observed state. 136
7.7 Release pipeline triggered by a version tag push. After the test gate passes, three parallel jobs produce platform binaries, Docker images, and a Helm chart. 141

8.1 Four-arm evaluation design, ordered as presented in this chapter: (1) proof-of-concept operational deployment on an internal developer platform, (2) controlled validation against a reference API suite, (3) large-scale validation against public APIs from APIs.guru, and (4) the agent-feedback experiment in which the validation report is the sole input to autonomous specification improvement. 145
8.2 Per-principle pass rates across the five evaluation APIs (strict mode). P001 achieves universal compliance; each remaining principle fails for exactly one API, reflecting the targeted defect injection (`bad-docs-api` for P002–P004/P008, `no-auth-api` for P005). 168

List of Tables

1.1 Mapping of research questions to thesis contributions. Direct support is marked ✓; supporting evidence is marked ˜. 29

2.1 Comparison of API quality approaches across six dimensions. . 44

3.1 Catalogue of the nine SDT principles. **Governing** = the axiom that primarily motivates the principle; **Impacted** = the remaining axioms, which every principle also satisfies (**C**ompleteness, **D**eterminism, **O**bservability). Severity reflects the source-of-truth constants in `driveby-cli/internal/types/report.go`. 62

3.2 Validation modes and the principles each mode evaluates. ✓ = full check set; ∼ = partial check set (detailed in the notes below); ✓° = runtime principle exercised outside the static checker registry. The registry's `ForMode()` method returns only the checkers listed for each mode. 64

4.1 The three abstraction layers of the system. 68

5.1 Package map of the DriveBy CLI. 75
5.2 Normalised specification types in the `spec` package. 79
5.3 Principle checker implementations. 86
5.4 CLI command structure. 92
5.5 CLI exit codes. 93

6.1 Representative configuration parameters and their two-tier resolution. 113
6.2 Parameters extracted from webhook payload by the Sensor. . . 115

7.1 Control theory mapping to DriveBy quality gate components. 134

8.1 Proof-of-concept configuration: gate checks per application per environment, reproduced from the `XSDLC` manifests in `kubernetes/examples/`. All five applications share the dev → staging → prod chain and carry `autoMerge:false` on the production transition; differences are confined to the gate checks. . . . 147
8.2 Activity volume captured at PoC end across all five applications. Sources: `results/cluster/all-*.json`, `results/cluster/<app>/*`. . . . 148
8.3 Operational edge-case test results on `private.novelcore.org`. ✓ = scenario reproduced and behaved as expected; n/r = scenario not reproducible under the deployed configuration (not a failure). . . . 151
8.4 CI/CD integration metrics for the DriveBy framework. Latency rows report values observed during the PoC; size rows are properties of the released artefacts. . . . 152
8.5 Automated SDT pipeline versus the manual equivalent it replaces. Automated values are measured in the PoC; manual values are engineering estimates (no manual control arm was run). . . . 153
8.6 Baseline validation results for `non-critical-api` across validation modes. . . . 155
8.7 Defect injection matrix: each injected defect targets one principle. 156
8.8 Per-check defect-injection results for `non-critical-api`. Detection means the targeted principle either (a) transitions from passing to failing, or (b) reports a new failed check that was passing in the baseline. . . . 157
8.9 Combination defect-injection results: simultaneous mutations on the `non-critical-api` baseline. . . . 157
8.10 Evaluation API suite: each API targets a distinct defect class. 159
8.11 Empirically measured strict-mode outcomes per API (source: `results/local/<api>-validate-strict.json`). **P** = principle fully passed; **F** = at least one check failed. P006 and P007 are runtime principles not exercised by static validation; their column reports the runtime-layer outcome: "runtime pass" = reached the runtime layer and passed, "runtime fail" = reached the runtime layer and failed, "–" = blocked statically before the runtime layer was reached. . . . 160

8.12 Local validation results in strict mode (6 principles). ✓ = pass, ✗ = fail. Severity: **C** = critical, **W** = warning. Critical-failure counts reflect the severity assignment in which P001–P005 are all critical (Section 3.4). 161
8.13 Staging gate results for all five evaluation APIs. 162
8.14 Production gate results. "Not reached" indicates the staging gate failed; the production gates configured for those APIs (Table 8.1) were therefore never exercised. 162
8.15 Layered validation defence: each layer catches defects invisible to prior layers. 163
8.16 Per-principle pass rates across the 50-API APIs.guru sample (strict mode). Bucket thresholds: high $\geq$ 80%, medium 30–79%, low $<$ 30%. 166
8.17 Strict-mode score distribution across the 50-API sample. Score = number of principles fully passing (out of six evaluated). . . 167
8.18 Per-principle pass rates on the 20-API Swagger 2.0 sample (strict mode). 169
8.19 SDT feedback experiment: per-principle outcomes for the Swagger Petstore specification, before and after a single agent-revision pass driven solely by the DriveBy validation report. ✓ = pass, ✗ = fail. 171

9.1 Authorship statistics across the entire project (52 commits, 3 April 2025 – 30 March 2026). 176
9.2 Approximate author effort allocation during the 12-day Phase B productisation sprint (author estimates; precise wall-clock data was not recorded). 177
9.3 CLAUDE.md hierarchy in the DriveBy monorepo. 181

Acronyms

Acronym	Definition
API	Application Programming Interface
ArgoCD	Argo Continuous Delivery
BDD	Behaviour-Driven Development
BYOCI	Bring Your Own CI
CI	Continuous Integration
CD	Continuous Delivery
CLI	Command-Line Interface
CR	Custom Resource
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DDD	Domain-Driven Design
DDT	Documentation-Driven Testing (the framework's earlier name; superseded by SDT)
DNS	Domain Name System
GHCR	GitHub Container Registry
GitOps	Git-based operations model for declarative continuous delivery
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IDP	Internal Developer Platform
JSON	JavaScript Object Notation
k8s	Kubernetes

Acronym	Definition
KubeCore	Novelcore's internal Kubernetes platform layer (Crossplane operator + compositions)
NFR	Non-Functional Requirement
OCI	Open Container Initiative
OpenAPI	The OpenAPI Specification (a proper name, not an acronym; formerly Swagger)
OWASP	Open Worldwide Application Security Project
P001–P009	SDT Validation Principles 1 through 9
PBT	Property-Based Testing
PoC	Proof of Concept
PR	Pull Request
RBAC	Role-Based Access Control
REST	Representational State Transfer
RQ	Research Question
SCM	Source Code Management
SDT	Specification-Driven Testing
SLA	Service Level Agreement
SSADM	Structured Systems Analysis and Design Method
TDD	Test-Driven Development
TLS	Transport Layer Security
URL	Uniform Resource Locator
XRD	Composite Resource Definition (Crossplane)
XSDLC	Crossplane Software Development Lifecycle composition
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

Modern software systems are built on APIs. They connect microservices, power mobile applications, expose business logic to partners, and increasingly serve as the interface through which autonomous agents interact with the world. Yet the quality assurance practices around these interfaces remain stubbornly manual, disconnected from the declarative pipelines that deploy the very services they test.

This thesis introduces Specification-Driven Testing (SDT)[1,2], a conceptual framework that treats the API specification not merely as documentation, but as the single source of truth from which validation rules, test cases, and quality gates are derived automatically. We present the theoretical foundations of SDT—three axioms grounding nine validation principles—together with an open-source implementation called DriveBy, a declarative Crossplane custom resource (XSDLC) that realises SDT quality gates inside GitOps delivery pipelines, and an empirical evaluation demonstrating its effectiveness across both controlled and large-scale settings. We position SDT explicitly as a conceptual framework—a strong theoretical core for a future formal methodology—rather than as a complete software-testing methodology in its own right; the elements that would be required to elevate it to a full

[1]An earlier formulation of this work used the name *Documentation-Driven Testing* (DDT); the name *Specification-Driven Testing* is adopted here as more precise, reflecting that the framework operates on a formal API specification (OpenAPI), not on documentation in general.

[2]Throughout the thesis, "API" refers specifically to *HTTP-based RESTful APIs* described by the OpenAPI specification (versions 3.0.x and 3.1.0), with limited treatment of Swagger 2.0 via the conversion adapter discussed in Chapter 5. The framework's axioms are stated at a level of generality that admits extension to other API styles—GraphQL introspection schemas, gRPC service definitions, AsyncAPI for event-driven systems—but those extensions are scoped as future work (Chapter 11). Where the text uses "API" without qualification, the RESTful + OpenAPI scope is implied.

methodology (test derivation procedure, execution and failure semantics, lifecycle governance) are identified as future work in Chapter 11.

1.1 The Cost of Manual Operations

1.1.1 The Talent Drain

Engineering talent is the scarcest resource in software organisations: the 2024 Stack Overflow Developer Survey reports that demand for software engineers continues to outpace supply [70]. Yet a remarkable share of their time goes not to building software but to verifying it by hand; the Postman State of the API Report confirms that manual testing remains the dominant API quality assurance practice [57].

The consequences compound. Forsgren, Humble, and Kim demonstrate that high-performing teams distinguish themselves through automated testing, deployment, and feedback loops, and that organisations failing to automate their quality gates systematically underperform on deployment frequency, lead time, change failure rate, and mean time to recovery [29]. The question is not whether to automate API quality assurance, but how.

1.1.2 Documentation Drift as a Systemic Failure

We define *documentation drift* as the progressive divergence between an API's specification and its actual implementation. Drift is not a one-time event but a continuous process: each deployment that modifies behaviour without updating the specification widens the gap. Over time, the specification becomes a historical artefact—accurate at some past commit, unreliable at the current one.

The SmartBear State of API Report identifies documentation quality as the single most important factor in API adoption, yet consistently finds that a majority of API specifications are incomplete, outdated, or both [67]. In most organisations, specification updates are a manual after-the-fact step enforced by nothing stronger than code review conventions; when deadlines compress, documentation is the first casualty.

Drift is not laziness but a systemic failure rooted in the separation of two artefacts—the specification and the implementation—that should be governed by a single source of truth. If validation derived from a complete specification ran on every change, drift would be detected at the point of introduction rather than discovered by a frustrated consumer weeks later.

What if the documentation itself could be the testing infrastructure?

1.2 The Shift Toward Declarative Everything

1.2.1 GitOps and the Declarative Promise

The past decade witnessed a fundamental shift in how software systems are operated. GitOps, formalised by the CNCF OpenGitOps working group, establishes four principles: declarative desired state, versioned and immutable storage, automated application of state, and continuous reconciliation [22]. Tools like Argo CD [9] implement these principles for Kubernetes [45]: an operator does not *do* things to a cluster but *declares* what it should look like, and a reconciliation loop makes it so.

Yet API quality assurance has not followed. Deployments and infrastructure provisioning are declarative, but the validation of the APIs those systems expose remains imperative: teams write tests by hand, run them in ad-hoc pipelines, and interpret results manually [57]. Declarative systems are tested with imperative processes.

1.2.2 Cloud-Native Infrastructure as Code

The declarative paradigm extends beyond application deployment. Terraform [38] pioneered infrastructure-as-code; Crossplane [26] turns the Kubernetes API server into a universal control plane that provisions databases, message queues, DNS records, and entire cloud environments through declarative manifests. The Cloud Native Computing Foundation's annual survey reports accelerating adoption of this model [18], whose philosophy is consistent: if you can describe it, you can automate it; if you can automate it, you can reconcile it; if you can reconcile it, drift disappears.

This philosophy has transformed infrastructure and deployment but not yet API quality assurance. The gap is not technological—OpenAPI specifications already describe APIs declaratively—but methodological: no widely adopted framework treats the specification as the source from which validation is derived, executed, and reconciled automatically.

1.2.3 The Agentic Development Frontier

A parallel development accelerates the urgency. AI-assisted software engineering tools—from code completion systems like GitHub Copilot [56] to autonomous agents capable of executing multi-step development workflows—are reshaping how software is built. These tools depend on structured, machine-readable inputs. An agent cannot reliably interpret a Confluence page or a Slack thread; it requires formal specifications with well-defined

semantics. This observation yields an explicit requirement for any quality-assurance methodology intended to operate in this setting: it must produce structured, specification-grounded quality signals that agents can consume and act upon without human interpretation.

SDT fulfils this requirement: its validation results are structured, machine-readable quality signals tied directly to the specification, which a CI pipeline can gate on, a reconciliation loop can consume, and an autonomous agent can reason about. We do not claim that agentic development is mature today; we observe that the trajectory is clear, and that frameworks producing specification-derived quality signals are inherently better positioned for it than approaches producing unstructured, human-authored test reports.

1.3 Structured Knowledge vs. Tribal Knowledge

1.3.1 The API Specification as Structured Knowledge

Every software organisation accumulates two kinds of knowledge about its APIs. The first is *tribal knowledge*: scattered across Slack messages, meeting notes, and the memories of senior engineers—rich, nuanced, and almost entirely inaccessible to automation. The second is *structured knowledge*: versioned, diffable, machine-readable, and testable. An OpenAPI specification [51] is the canonical example, describing endpoints, parameters, request bodies, response schemas, authentication mechanisms, and error codes in a format that both humans and machines can consume.

The distinction matters because automation operates on structured knowledge: a validation engine cannot derive test cases from a Slack conversation. The degree to which API quality assurance can be automated is bounded by the degree to which the API contract is captured in a structured, machine-readable form.

1.3.2 Platform Engineering and Knowledge Codification

The structured-versus-tribal distinction has organisational consequences beyond automation. Nonaka and Takeuchi's theory of organisational knowledge creation [48] distinguishes *tacit* knowledge (embodied in individual expertise) from *explicit* knowledge (codified in artefacts), and identifies *externalization*—converting tacit into explicit form so that it survives personnel transitions—as the central challenge of knowledge management. In software organisations, the CNCF Platforms White Paper [19] identifies internal developer platforms as the primary externalization mechanism, and Skelton and Pais [64] argue that

platform teams reduce cognitive load on stream-aligned teams by codifying collective expertise into well-documented, curated interfaces.

API specifications occupy a unique position in this landscape: they are simultaneously a technical contract, a documentation artefact, and a knowledge repository encoding decisions about authentication flows, error semantics, versioning strategy, and data models—but only if the specification is complete enough to include them.

SDT operationalizes this insight. A specification that passes all nine SDT principles is not merely "valid" in the schema sense; it is *onboarding-complete*: a new engineer can understand the API's contract, security model, error behaviour, and versioning strategy by reading the specification alone. The validation report identifies precisely where knowledge gaps remain—missing descriptions, undocumented error responses, absent security definitions—converting the diffuse problem of "incomplete documentation" into a concrete, actionable checklist. When the engineer who designed the authentication flow leaves, the knowledge persists in a specification whose completeness is continuously validated on every pipeline run.

The same completeness serves autonomous engineering agents, to whom tribal knowledge is *entirely* inaccessible: an SDT-validated specification is exactly the structured, machine-readable contract that agent architectures require to reason about API interactions [77]. SDT does not predict when agents will routinely maintain APIs; it ensures the knowledge substrate is machine-readable today, so that it is agent-ready when the tooling matures.

1.3.3 Why Auto-Generated Specifications Matter

SDT rests on an assumption of specification completeness: the specification must fully describe the API contract. This is increasingly achievable: frameworks such as FastAPI [60], Spring Boot, and Express with swagger-jsdoc auto-generate OpenAPI specifications directly from annotated source code, making the specification a byproduct of the implementation rather than a separate artefact to maintain.

However, auto-generated specifications are structurally correct but semantically incomplete: a framework generates endpoint paths, parameter types, and response status codes, but not meaningful descriptions, usage examples, or business-logic constraints. This gap is precisely what SDT detects. Its principles assess not only whether the specification parses, but whether it contains the information a consumer needs—descriptions on every operation, examples on every schema, documented error responses, explicit security definitions. SDT does not replace auto-generation; it audits the output and identifies where human enrichment is still required.

1.4 Problem Statement

API quality assurance has not undergone the declarative transformation that has reshaped infrastructure provisioning, application deployment, and continuous delivery. While the rest of the software delivery pipeline converges toward Git-versioned, automatically reconciled desired state, API validation remains a manual, imperative process: teams write tests by hand, maintain them separately from the specification, and run them in pipelines disconnected from the declarative workflows that govern everything else.

The consequences are measurable. Documentation drift accumulates undetected. Quality regressions surface in production rather than in CI. Engineering time flows toward manual verification rather than automation. Existing tools address fragments of the problem—Spectral [71] lints specification style, Schemathesis [28] generates property-based tests from schemas, EvoMaster [7] evolves system-level test suites from API schemas, Pact [54] validates consumer-provider contracts—but none offers a unified *framework* that derives comprehensive validation from the specification alone, grounded in formal principles, and designed for integration into GitOps pipelines.

The opportunity is concrete. OpenAPI specifications already contain the information needed to validate compliance, documentation quality, schema correctness, error handling, security definitions, and versioning strategy. What is missing is not data but a principled framework: one that maps specification content to validation rules, executes those rules deterministically, and produces structured, observable results suitable for automated quality gates.

1.5 Research Questions

This thesis investigates the following research questions:

RQ1: *Can a conceptual framework grounded in specification completeness, behavioural determinism, and operational observability demonstrate the feasibility of automated API quality assurance that can operate without dedicated testers in the promotion pipeline?*

We formalise SDT as three axioms mapped to nine validation principles and evaluate whether the resulting automated checks detect the same categories of defects that manual review identifies—with zero test authorship, zero human interpretation, and zero dedicated QA personnel. This question is addressed in Chapters 3 and 8.

RQ2: *To what extent can validation rules, test cases, and quality gates be derived automatically from the OpenAPI specification together with*

declared operational policy (the XSDLC resource), and does the resulting automation produce actionable, machine-readable signals suitable for both human engineers and autonomous agents?

We implement the DriveBy framework, which takes an OpenAPI specification as sole input and produces structured validation reports with per-principle pass/fail status, severity classifications, and suggested remediations. We evaluate this against both a controlled reference API and a large-scale public API dataset, and an agent-feedback experiment demonstrates that the reports' remediation signals are actionable by an autonomous agent. This question is addressed in Chapters 5 and 8.

RQ3: *How effectively does specification-driven quality assurance integrate into cloud-native GitOps pipelines, and can a single declarative custom resource replace the manual provisioning of quality gate infrastructure?*

We design XSDLC, a Crossplane custom resource definition that generates a complete quality-gated delivery pipeline—ArgoCD Applications, Argo Workflows, Argo Events infrastructure, branch protection rules, and GitOps Promoter resources—from approximately 35 lines of YAML in under 15 seconds, replacing hours of manual ClickOps. As a proof of concept, we deploy this on an operational cluster alongside existing infrastructure, validating surgical coexistence and removal. This question is addressed in Chapters 6, 7, and 8.

RQ4: *Can quality-gated delivery pipelines be expressed as Kubernetes-native ontological specifications that operators and autonomous agents can discover, inspect, and manage as composable building blocks within an internal developer platform?*

We demonstrate that the XSDLC Custom Resource Definition (CRD)[3] functions as a machine-readable, structurally typed, discoverable specification—not merely a tool configuration—that integrates into Novelcore's Internal Developer Platform (IDP)[4] alongside database

[3]A *Custom Resource Definition* (CRD) is the Kubernetes extension mechanism by which a user-defined resource type—with its own schema, validation, and API endpoint—is registered with the Kubernetes API server, so that instances of it (custom resources) can be created, inspected, and reconciled like any built-in resource.

[4]An *Internal Developer Platform* (IDP) is the Kubernetes-native abstraction layer that an organisation builds on top of its raw cluster infrastructure to give application teams a curated, declarative interface for provisioning the resources they need (databases, networking, observability, delivery pipelines). The Cloud Native Computing Foundation's Platform Engineering working group documents the pattern; the Crossplane operator and Backstage are typical building blocks. Novelcore's IDP, branded internally as *KubeCore*, is

operators, networking compositions, and observability stacks, all following the same Crossplane pattern. This question is addressed in Chapters 6 and 11.

1.6 Thesis Contributions

This thesis presents a proof-of-concept system for specification-driven API quality assurance—the DriveBy validation framework together with the XSDLC declarative pipeline that embeds it in GitOps delivery—and makes six contributions. A central scope boundary applies throughout: SDT does not replace all testing—it replaces testing at the promotion gate, while developers' own unit and CI tests remain outside its boundary, a separation formalised as the Bring-Your-Own-CI (BYOCI) principle detailed in Chapter 7. Section 1.6.1 provides an explicit mapping between these contributions and the four research questions of Section 1.5.

1. **A Conceptual Framework for Specification-Driven API Validation.** We define Specification-Driven Testing (SDT) as a conceptual framework—a strong theoretical core for a future formal methodology—built on three framework-wide axioms (Completeness, Determinism, Observability) that every validation must jointly satisfy, and nine validation principles (P001–P009) that operationalize the framework in practice. SDT provides a reusable, specification-agnostic basis for deriving API quality gates from a formal API contract. We claim it as a conceptual framework rather than a complete software-testing methodology: the elements that would elevate it to a full methodology—a formal test derivation procedure, execution and failure semantics, and lifecycle governance—are scoped as future work (see Chapter 11).

2. **The DriveBy Framework.** We present DriveBy [74], an open-source command-line tool that implements SDT validation for OpenAPI specifications. DriveBy supports OpenAPI 3.x and Swagger 2.x, operates in multiple validation modes (minimal, strict, test-only, test-ready), and produces structured validation reports suitable for CI/CD integration.

3. **The XSDLC Declarative Pipeline.** We design the XSDLC Crossplane custom resource definition that generates a complete quality-gated delivery pipeline from a single Custom Resource (CR) ($\sim$35 lines of YAML $\rightarrow$ 45–50 Kubernetes objects in 15 seconds). XSDLC replaces

built on Crossplane compositions and is the deployment environment for the operational evaluation in Chapter 8.

hours of manual ClickOps with a declarative, version-controlled, self-healing specification that can be surgically installed into and removed from existing Kubernetes clusters without affecting pre-existing infrastructure.

4. **XSDLC as Ontological Specification.** We demonstrate that the XSDLC CRD functions as a Kubernetes-native ontological artefact [37]—discoverable, inspectable, and manageable by operators and autonomous agents as a composable building block within an IDP. This contribution reframes quality-gated delivery as a platform capability expressed through structured, machine-readable specifications rather than imperative tooling.

5. **Empirical Evaluation.** We evaluate SDT through four complementary experiments: a controlled evaluation against a reference API, an agent-driven large-scale evaluation against public APIs from APIs.guru [6], a proof-of-concept deployment on Novelcore's IDP demonstrating the feasibility of zero-tester quality assurance and declarative load testing contracts, and an agent-feedback experiment in which an AI agent iteratively improves an OpenAPI specification using DriveBy's validation reports as its sole feedback signal.

6. **An Agent-Driven Development Framework.** We document the AI-assisted development framework used to build DriveBy, centered on the CLAUDE.md knowledge protocol—a hierarchical system of machine-first documentation files that provides AI agents with persistent, structured context for cross-session collaboration. We claim this as an enabling framework that operates within an existing software development lifecycle, not as a standalone methodology in itself. The framework was exercised throughout the 12-day Phase B agent-partnership window—the second of the two development phases distinguished in Chapter 9, during which an AI agent acted as the primary implementer under human direction—and is documented quantitatively in that chapter. The same window demonstrates that SDT's structured output can be directly consumed by AI agents for automated evaluation and specification remediation.

1.6.1 Research Questions and Contributions Mapping

Table 1.1 traces each research question to the contributions that address it, and to the chapter where the evidence is presented. A check mark (✓)

indicates that the contribution directly addresses the research question; a tilde (~) indicates partial or supporting evidence.

Table 1.1: Mapping of research questions to thesis contributions. Direct support is marked ✓; supporting evidence is marked ~.

Question	C1	C2	C3	C4	C5	C6	Primary chapters
RQ1	✓	✓			✓	~	Ch. 3, 8
RQ2	~	✓			✓	~	Ch. 5, 8
RQ3	~	~	✓		✓		Ch. 6, 7, 8
RQ4			✓	✓	~		Ch. 6, 11

Contribution legend: **C1** SDT conceptual framework; **C2** DriveBy implementation; **C3** XSDLC declarative pipeline; **C4** XSDLC as ontological specification; **C5** empirical evaluation; **C6** agent-driven development framework.

1.7 A Technology Showcase

Beyond the conceptual contributions enumerated above, this thesis serves as a reference implementation of how contemporary software is built, automated, and delivered: Kubernetes-native automation through Crossplane custom resource definitions, event-driven orchestration through Argo Events and Argo Workflows, multi-environment promotion through the GitOps Promoter, three CI/CD pipelines producing multi-platform binaries, OCI container images, and Helm charts, and AI-assisted development through agentic workflows with structured knowledge protocols. These technologies are the medium through which SDT becomes operational—a quality-assurance framework that cannot be deployed declaratively, triggered automatically, and promoted through environment gates remains an academic exercise. The implementation is a concept validation—a working prototype that demonstrates feasibility and architectural viability, not a production-hardened product.

1.8 Thesis Structure

The remainder of this thesis is organised as follows:

Chapter 2 surveys related work and identifies seven research gaps that position SDT within the existing landscape.

Chapter 3 presents the SDT framework: axioms, principles, their parallel-application semantics, and the four validation modes.

Chapter 4 establishes the system boundary, the three-layer model, and the architectural evolution from CLI tool to custom resource.

Chapter 5 analyses the DriveBy CLI's internal architecture, from the `APISpec` abstraction to report generation.

Chapter 6 presents the Kubernetes architecture: the XSDLC XRD, composition pipeline, event infrastructure, and multi-check DAG.

Chapter 7 details the GitOps pipeline: the promotion-driven quality gate flow, the two-repo BYOCI architecture, and the gate as a closed-loop control system.

Chapter 8 presents the four-arm empirical evaluation: controlled (`non-critical-api`), large-scale (APIs.guru), operational (Novelcore's IDP), and agent-feedback.

Chapter 9 examines the AI-assisted development method (CLAUDE.md knowledge protocol, agent memory, agent-driven evaluation and thesis writing) and quantifies the author's own degree of contribution to the code and to this report.

Chapter 10 discusses the results and revisits the research questions with evidence from the evaluation.

Chapter 11 concludes with contributions, the ontological framing of XSDLC, limitations, and future work.

Chapter 2

Related Work

This chapter surveys the landscape of API quality assurance, examining the tools, methodologies, and paradigms that inform Specification-Driven Testing. We organise the discussion into five domains—the OpenAPI ecosystem, API testing and verification, GitOps and continuous delivery, infrastructure-as-code, and AI-assisted software engineering—before synthesizing the identified gaps into a unified argument for SDT's contribution.

2.1 The OpenAPI Ecosystem

2.1.1 The OpenAPI Specification

The OpenAPI Specification [51] provides a standard, language-agnostic interface description for HTTP APIs. Originally released as Swagger 2.0 in 2014, it was donated to the Linux Foundation's OpenAPI Initiative in 2015 and has since evolved through three major 3.x revisions [49]. OpenAPI 3.0 (2017) introduced components, multiple server definitions, and improved content negotiation. OpenAPI 3.1 (2021) aligned the schema model with JSON Schema 2020-12, enabling full JSON Schema compatibility including keywords like `if`/`then`/`else` and `prefixItems`. OpenAPI 3.2 (released by the OpenAPI Initiative in 2025) adds enhanced data-modelling primitives, expanded HTTP method support, and affordances for patterns such as streaming; the present work targets 3.0/3.1 specifications, with 3.2 support flagged as straightforward future work in Chapter 11. A specification document is a tree of top-level sections—`openapi`, `info`, `servers`, `paths`, and reusable `components`—to which the validation principles of Chapter 3 map directly.

Adoption is broad. The Postman State of the API Report identifies OpenAPI as the most-used API specification format [57], and SmartBear's survey

confirms that organisations across industries rely on it for documentation, code generation, and testing [67].

However, the specification is intentionally permissive. A minimal OpenAPI document requires only a title, version, and at least one path—no descriptions, no examples, no error responses, no security definitions. This permissiveness makes adoption easy but quality optional: a specification can be *valid* according to the schema yet *useless* as documentation. This gap between validity and quality is precisely where SDT operates.

2.1.2 OpenAPI Production Workflows: Code-First and Design-First

Two competing workflows produce OpenAPI specifications. In code-first workflows, frameworks generate the specification from annotated source code. FastAPI [60] generates OpenAPI 3.1 from Python type hints and decorator metadata. springdoc-openapi [69] does the same for Spring Boot applications. The specification is a byproduct of implementation—always structurally synchronised with the code but often semantically thin, lacking descriptions, examples, and documentation beyond what the framework infers from type signatures.

In design-first workflows, teams author the specification before writing implementation code. Platforms like Stoplight [72] provide collaborative visual editors, and the specification drives code generation, mock servers, and documentation. Design-first workflows produce richer specifications but face a synchronisation problem in the opposite direction: as the implementation evolves, the specification must be manually updated.

Research prototypes attempt to bridge the two: Smardas and Kritikos generate richer OpenAPI specifications directly from source code [65], narrowing the semantic gap of framework-driven generation. The underlying problem persists, however: a generated specification is only as correct as the implementation it derives from—an erroneous implementation yields a faithful specification of the wrong contract.

Neither workflow ensures ongoing specification completeness: code-first inherits the implementation's structural shapes without semantic detail, design-first drifts from the running system—a quality gap no existing tool systematically addresses.

2.2 API Testing: Foundations and Taxonomy

Golmohammadi, Zhang, and Arcuri provide a comprehensive survey of RESTful API testing techniques, cataloguing approaches ranging from random testing to model-based generation [35]. We organise these approaches into three categories by primary mechanism: (1) contract testing and (2) property-based/fuzz testing, both of which exercise the running API, and (3) specification linting—not testing but a complementary static-verification technique operating on the document alone.

2.2.1 Contract Testing

Contract testing verifies that interacting services agree on the shape and semantics of their communication. Two sub-types exist, distinguished by who authors the contract: in *provider-driven* contract testing the provider's own API specification is the contract that the running implementation must honour; in *consumer-driven* contract testing each consumer publishes the expectations it depends on and the provider proves compliance.

Provider-Driven Contract Testing

Dredd [3], one of the earliest specification-driven testing tools, reads an API Blueprint or OpenAPI document, generates HTTP requests for each documented endpoint, and compares responses against the declared schemas. Although influential in establishing the concept of testing *from* a specification, Dredd's development has slowed and its OpenAPI 3.x support remains limited.

Portman [5] converts OpenAPI specifications into Postman collections with auto-generated test assertions. Microcks [47], a CNCF Sandbox project, generates mock endpoints from OpenAPI specifications and validates live API responses against the same documents.

These tools treat the specification as the source of test cases and verify that runtime behaviour matches the declared contract. This is valuable but insufficient: they test *from* the specification but not *the specification itself*. Dredd confirms that `GET /users` returns a 200 response matching the declared schema, but does not assess the specification's completeness or design quality—documented error responses, operation descriptions, security scheme definitions, versioning and governance conventions.

Consumer-Driven Contract Testing

Robinson introduced the consumer-driven contracts pattern in 2006, proposing that API consumers publish expectations that providers must satisfy [62]—

inverting the traditional testing relationship in which the provider defines what is correct.

Pact [54] is the most widely adopted implementation of consumer-driven contracts. Consumers record interactions during local test runs against a framework-supplied mock provider—the real provider API is never invoked at this stage—and the recordings become contract files that the provider verifies independently. Pact supports multiple languages and offers a broker for managing contracts across teams. Spring Cloud Contract [75] provides a similar capability within the Spring ecosystem, generating test stubs[1] from contract definitions written in Groovy or YAML.

Consumer-driven contract testing excels at preventing integration breakage between known consumer-provider pairs. However, it requires active consumer participation: each consumer must author and maintain its contract. In organisations with dozens of consumers—or with public APIs consumed by unknown third parties—this requirement becomes impractical. More fundamentally, contract testing validates the *interface agreement* between two parties but does not assess the *quality* of the underlying specification: a contract can pass while documentation remains incomplete, error responses undocumented, and security definitions absent.

2.2.2 Property-Based and Fuzz Testing

Property-based testing (PBT) generalises example-based testing by specifying properties that must hold for all inputs, then generating random inputs to search for violations. Claessen and Hughes introduced QuickCheck [17] for Haskell, demonstrating that random generation with *shrinking*—the automatic minimisation of a failing input to its smallest still-failing form—could discover subtle bugs that hand-crafted test cases missed. In the API-testing context of this thesis, fuzzing is understood as the input-generation mechanism PBT employs: schema-derived random request payloads replayed against the API to surface schema violations.

Schemathesis [28] applies property-based testing to API schemas: given an OpenAPI or GraphQL specification, it generates randomized HTTP requests and verifies that responses conform to the declared schemas, automatically detecting server errors, schema mismatches, and undocumented status codes.

RESTler [14], developed at Microsoft Research, extends fuzzing to stateful API interactions. RESTler infers producer-consumer dependencies between endpoints—for instance, that creating a resource returns an identifier needed

[1] A *test stub* is a lightweight stand-in for a real service that returns canned responses, allowing one side of an interaction to be tested without deploying the other.

by subsequent requests—and generates sequences of API calls that exercise these dependencies, an approach that has discovered crashes and security violations in production services at Microsoft.

EvoMaster [7] pursues a related goal through search-based software testing, applying evolutionary algorithms to OpenAPI-described services to evolve HTTP call sequences that maximise coverage and have repeatedly uncovered faults in open-source REST APIs.

Both approaches test the *robustness* of the API implementation against its schema, not whether the specification itself is complete, well-documented, and fit for purpose: an API can pass all Schemathesis checks while its specification lacks descriptions, omits security definitions, and provides no examples for consumers.

2.2.3 Specification Linting and Static Analysis

Specification linters analyse API specifications without executing the API—static analysis rather than testing, surveyed here as a verification technique complementary to the testing approaches above—enforcing structural rules, naming conventions, and documentation standards through configurable rulesets.

Spectral [71], maintained by Stoplight, is the most established OpenAPI linter. It ships with a default ruleset covering common issues—missing descriptions, unused schemas, inconsistent naming—and supports custom rules written in JavaScript or JSON.

Vacuum [55] implements the same ruleset language as Spectral but executes validation significantly faster, suiting large specifications and speed-sensitive CI pipelines.

Redocly [61] combines linting with interactive documentation generation from the same source. SwaggerHub [66] provides a hosted platform integrating specification editing, linting, and collaboration. In current practice, these linters run as advisory checks—warnings in continuous-integration jobs and editor feedback—rather than as gates blocking promotion toward production.

Optic [52] focuses instead on API change detection, comparing specification versions to identify breaking changes, new endpoints, and modified schemas within pull request (PR) workflows.

These tools share a critical limitation: they operate purely at the static level. Spectral can verify that every operation has a description, but it cannot verify that the described behaviour matches the implementation. Static analysis addresses documentation *form* but not documentation *fidelity*.

2.3 GitOps and Continuous Delivery

2.3.1 GitOps Principles

GitOps, as formalised by the CNCF OpenGitOps working group [22], establishes four principles for operating software systems: (1) declarative desired state, (2) versioned and immutable storage in Git, (3) automated application of approved changes, and (4) continuous reconciliation ensuring the running system converges toward the declared state. These principles eliminate imperative drift: operators declare what should exist, and a reconciliation loop ensures it does.

The model is a control loop—desired state in Git, observed state in the cluster, a controller correcting the difference; Section 7.6 develops this control-theoretic framing in full.

GitOps transformed how organisations deploy software. Yet its reconciliation model has not, to the best of our knowledge, been systematically applied to API quality assurance: the survey of Golmohammadi et al. [35] catalogues no reconciliation-based approach. The omission is conspicuous because the API specification is itself a declaration of desired state—the contract the running API should satisfy—yet APIs in current practice neither converge toward their specifications nor are guarded against the specification drifting from the implementation it claims to describe. Adjacent capabilities exist—Optic [52] computes specification-version drift, APIClarity [4] reconstructs specifications from production traffic, 42Crunch [1] continuously audits OpenAPI security, and gateways such as Kong [42] enforce specification-derived policies at the edge—but none combines the three elements convergence requires: a declared target (a declarative principle catalogue), an observable deviation signal (machine-readable per-principle output), and an actuator responding to that signal (a reconciliation-style promotion gate within a GitOps pipeline); Section 2.4.4 develops this correspondence in full. Each existing tool covers a slice; SDT integrates the slices into a single specification-driven gate.

2.3.2 Continuous Delivery Tools

Argo CD [9] implements GitOps for Kubernetes. It watches Git repositories for changes, compares the desired state against the running cluster, and synchronises differences automatically or on approval. Flux [20], another CNCF project, provides similar capabilities with a different architectural approach, using custom controllers rather than a centralized application.

Argo Workflows [11] extends the Argo ecosystem with container-native workflow execution: pipelines are expressed as directed acyclic graphs (DAGs)

of container steps. Argo Events [10] adds event-driven triggering from Git pushes, webhooks, or message queues.

These tools provide the infrastructure for automated quality gates but do not prescribe what those gates should validate. A team can configure Argo Workflows to run Spectral on every pull request, but this integration is ad-hoc—the team must decide which tools to run, how to interpret results, and where to gate.

2.3.3 Quality Gates in CI/CD

Shift-left testing—moving validation earlier in the development lifecycle, as articulated by Smith [68]—is implemented in modern CI/CD pipelines through automated checks on every commit or pull request.

The DORA research program [36], building on Forsgren, Humble, and Kim [29], links superior delivery performance on four key metrics (deployment frequency, lead time for changes, change failure rate, mean time to recovery) to comprehensive automation, including automated testing and deployment.

Yet API-specific quality gates remain the exception rather than the norm. SmartBear's State of API Report consistently finds that API testing is the least automated aspect of the development lifecycle [67]. The tooling exists—linters, fuzzers, contract testers—but no unified methodology prescribes how to combine them into a comprehensive, automated quality gate that fits naturally into a GitOps workflow.

2.3.4 Release Engineering and Artefact Automation

Modern release engineering extends beyond testing into artefact production. GoReleaser [15] automates the production of multi-platform binaries, checksums, and GitHub Releases from a single tag push. OCI-compliant registries [39] serve not only container images[2] but also Helm charts, enabling `helm install` directly from OCI artefacts stored alongside the application images they describe. Docker Buildx [27] extends Docker builds to multi-platform images (amd64 and arm64) from a single pipeline.

These tools form the final link in the automation chain—commit to validated, multi-platform, registry-hosted artefacts—within which SDT's quality gate operates: validation results gate promotion, and promotion triggers release automation. Chapter 7 describes how DriveBy's own CI/CD pipelines compose these tools.

[2] A *container image* is a layered, immutable filesystem bundle packaging an application with its runtime dependencies, conforming to the Open Container Initiative (OCI) image specification.

2.3.5 Environment Promotion and Progressive Delivery

GitOps pipelines typically manage multiple environments with promotion gates between them; the GitOps Promoter [34] automates promotion by merging promotion pull requests when all commit-status gates pass, a mechanism Section 7.4.4 details.

Progressive delivery extends this model with canary releases, blue-green deployments, and feature flags. The key insight shared by progressive delivery and SDT is that promotion decisions should be automated, evidence-based, and reversible—consistent with the delivery-performance findings of Forsgren, Humble, and Kim [29]. SDT provides the evidence (structured validation results as commit status signals); the GitOps Promoter provides the automation (merge on success, block on failure); and the GitOps model provides reversibility (rollback by reverting a Git commit).

2.4 Infrastructure as Code and the Declarative Paradigm

2.4.1 Terraform and Infrastructure as Code

Terraform [38], created by HashiCorp, pioneered the declarative management of cloud infrastructure. Engineers declare the desired state of their infrastructure in configuration files; Terraform computes and applies a plan to reach that state, and its state file enables drift detection between declared and actual infrastructure.

Pulumi [59] extends this model by allowing infrastructure definitions in general-purpose programming languages rather than a domain-specific language. Both tools share the core insight: describing infrastructure declaratively and reconciling differences automatically eliminates an entire class of operational errors.

This pattern—declare desired state, detect drift, reconcile—is directly analogous to what SDT proposes for API quality: the specification declares the contract, the validation engine detects drift, and the promotion gate reconciles. Section 2.4.4 formalises this correspondence.

2.4.2 Crossplane: The Universal Control Plane

Crossplane [26] extends the Kubernetes API to manage external resources, turning the Kubernetes control plane into a universal control plane: teams declare databases, cloud services, and entire environments as Kubernetes

custom resources. Crossplane controllers watch these resources, provision the corresponding infrastructure through cloud provider APIs, and continuously reconcile the actual state against the declared state.

Crossplane is relevant to SDT in two ways. First, it demonstrates that the Kubernetes reconciliation model can govern resources beyond containers; SDT borrows this reconciliation mechanism, treating API quality assessments as a new class of governed resource. Second, it provides the infrastructure provisioning layer for the end-to-end pipeline described in Chapter 7.

2.4.3 Policy as Code

Open Policy Agent (OPA) [21] provides a general-purpose policy engine that evaluates structured data against declarative policies written in the Rego language. In Kubernetes environments, OPA Gatekeeper enforces admission policies—ensuring that deployed resources meet organisational standards for security, labelling, and resource limits.

Kyverno [46] offers a Kubernetes-native alternative, expressing policies as custom resources rather than in a separate language. Both tools implement the same pattern: declare what should be true about a resource, evaluate every resource against those declarations, and reject resources that violate policy.

The analogy to SDT is direct: where OPA asks "Does this Pod have resource limits?", SDT asks "Does this specification document its error responses?"—and where OPA and Kyverno reject non-conforming resources at admission time, an SDT quality gate rejects promotion of a non-conforming API.

2.4.4 Ontological State Reconciliation

The tools surveyed in this section—Terraform, Crossplane, OPA, Argo CD—share a common operational pattern that deserves explicit formalisation. Each tool maintains a *declared ontology*: a structured description of what entities should exist, what properties they should have, and what relationships should hold between them. Each tool continuously compares this declared ontology against *observed reality* and takes corrective action when the two diverge. We term this pattern *ontological state reconciliation*, following Gruber's definition of an ontology as "an explicit specification of a conceptualization" [37].

The pattern has three components. First, an *ontological artefact* declares the desired state in a structured, machine-readable format (Kubernetes manifests for Argo CD, configuration and state for Terraform, Rego policies for OPA). Second, a *reconciliation agent* observes the actual state of the governed

system. Third, a *correction mechanism* drives the actual state toward the declared state, either by modifying the system (Argo CD sync, Terraform apply) or by rejecting non-conforming changes (OPA admission control).

SDT instantiates this pattern for API quality. The ontological artefact is the OpenAPI specification—a structured declaration of the API's entities (schemas), operations (paths and methods), constraints, security mechanisms, and documented error conditions. The reconciliation agent is the DriveBy validation engine, which observes the specification's actual state of completeness and the API's runtime behaviour. The governed resource is therefore dual: the static principles (P001–P005, P008, P009) govern the specification document, while the runtime principles (P006, P007) govern the running implementation. The correction mechanism is the GitOps quality gate, which blocks promotion when the declared ontology and the observed reality diverge—and the forced remediation is correspondingly dual: a static finding is corrected by enriching the specification, a runtime finding (an undocumented status code, a breached latency threshold) by fixing the implementation's source code or capacity, not the document.

The XSDLC custom resource (Chapter 6) extends the same family one level up: its declared ontology is the delivery pipeline itself—the environments that exist, the order promotion traverses them, and the non-functional constraints (load profile, latency, success rate) imposed on the runtime API at each gate.

This framing illuminates why SDT's axiomatic foundation matters. The Axiom of Completeness requires that the ontological artefact—the specification—is expressive enough to serve as the sole input to reconciliation. The Axiom of Determinism requires that the reconciliation agent produces consistent observations. The Axiom of Observability requires that the correction mechanism receives structured, actionable signals. The three axioms are not arbitrary; they are the necessary conditions for ontological state reconciliation to function.

The framing also clarifies what distinguishes SDT from existing API tools: linters validate the ontological artefact without reconciling it against runtime reality, and fuzzers test the governed system without assessing the artefact's completeness, whereas SDT performs full ontological state reconciliation—validating the artefact's completeness, verifying the system's conformance, and producing structured signals for automated correction.

Finally, the reconciliation agent need not be a deterministic controller: autonomous software engineering agents [77] can parse SDT's machine-readable per-principle output—principle IDs, pass/fail status, check failures, suggested remediations—and generate corrective pull requests, extending the loop to specification quality convergence driven by AI agents [76] (Section 2.5).

2.4.5 Platform Engineering and Developer Experience

A parallel development in software engineering practice provides additional context for SDT's contribution. Platform engineering—the discipline of building and maintaining internal developer platforms—has emerged as a response to the cognitive overload imposed by cloud-native complexity [19]. The Humanitec State of Platform Engineering Report documents rapid adoption of self-service platforms that abstract infrastructure and operational concerns behind curated interfaces [40].

Skelton and Pais formalise this trend in *Team Topologies* [64], proposing four team types—stream-aligned, platform, enabling, and complicated-subsystem—where the platform team's core function is to reduce stream-aligned teams' cognitive load by encoding operational knowledge into the platform itself.

This model has a direct implication for API quality. In a platform-oriented organisation, the API specification is a platform artefact. It is the interface through which stream-aligned teams consume and provide capabilities. If the specification is incomplete—missing descriptions, undocumented errors, absent security definitions—it imposes cognitive load on every consumer, who must seek tribal knowledge to fill the gaps. Conway's Law [24] predicts that these knowledge gaps will manifest as integration friction between teams.

SDT addresses this by ensuring that the specification meets a minimum knowledge threshold: the validation principles (Chapter 3) map directly to the information needs of platform consumers. A specification that passes SDT's strict mode is, by design, a self-service artefact—a new team can integrate with the API using only the specification, without consulting the providing team. Documentation quality alone does not complete the guarantee: a drifted specification frustrates consumers as surely as an incomplete one, so the runtime principles (P006, P007) guard the specification–implementation gap—the self-service property holds only while the running API conforms to its document.

This positions SDT as a platform capability: a quality gate that platform teams deploy to ensure that all APIs within the organisation meet the documentation standard required for self-service consumption. The validation report identifies precisely where specifications fall short, enabling targeted remediation rather than blanket documentation mandates.

Platform consumers increasingly include AI agents that integrate services on behalf of development teams [76]. An agent has strictly less tolerance for ambiguity than a human: it cannot infer intent from endpoint naming, ask a colleague what an undocumented 422 response means, or improvise authentication from incomplete security definitions. SDT's completeness

principles (P001–P004) make validated specifications agent-ready platform artefacts—a quality bar that transitions from documentation best practice to functional prerequisite as agents become routine consumers.

2.5 AI-Assisted Software Engineering

The emergence of AI-assisted software engineering tools introduces a qualitatively new class of consumer for structured API quality signals. We distinguish three levels of AI assistance in software development.

Code completion. GitHub Copilot and similar tools predict the next lines of code from context. Peng et al. demonstrate measurable productivity gains from code completion [56], but these tools operate at the syntactic level: they do not reason about API contracts, quality signals, or deployment pipelines—consumers of documentation as context, not of structured validation output.

Autonomous coding agents. SWE-agent [77] and Devin [23] represent a step beyond completion. These systems accept a task description, explore a codebase, formulate a plan, implement changes across multiple files, and submit the result for review; Wang et al. survey such agents comprehensively [76]. These agents require structured, machine-readable inputs—they cannot parse Slack conversations or infer intent from naming conventions.

Agentic development workflows. A further development is the emergence of structured knowledge protocols that bridge human intent and agent execution. The CLAUDE.md convention—a hierarchical system of project-level and directory-level instruction files—provides agents with persistent, versioned context about project structure, coding conventions, and architectural decisions. Unlike READMEs (written for humans) or configuration files (written for specific tools), CLAUDE.md files are written for agents: machine-first documentation that enables cross-session continuity and principled delegation of development tasks.

SDT's structured output is naturally positioned for the latter two levels of assistance: code-completion tools consume documentation only as ambient context, whereas autonomous agents and agentic workflows can act on machine-readable diagnostics directly. A validation report with per-principle pass/fail status, specific check failures, and suggested remediations is an instruction set such an agent can parse, reason about, and act on. The convergence of specification-driven validation (SDT) and agent-driven development suggests a feedback loop in which agents validate, diagnose, remediate, and re-validate API specifications autonomously. Chapter 9 explores this convergence in detail, drawing on the experience of using AI-assisted development throughout the

construction of this system and this thesis; the loop is evaluated empirically in Section 8.6, where an autonomous agent raised the canonical Swagger Petstore specification [50] from 1/6 to 4/6 passing principles using the SDT validation report as its sole feedback signal.

2.6 Synthesis and Research Gap

2.6.1 Taxonomy of API Quality Approaches

Table 2.1 presents a comparative analysis of existing API quality tools across six dimensions that together characterise a comprehensive quality assurance methodology. *Static analysis* denotes the ability to validate specifications without a running API. *Runtime testing* denotes the ability to test a live API instance. *Spec quality assessment* denotes whether the tool evaluates the completeness and documentation quality of the specification itself, rather than merely consuming it as input. *Formal methodology* denotes whether the tool is grounded in explicitly stated axioms or principles. *GitOps-native* denotes whether the tool is designed for integration into declarative, reconciliation-based pipelines. *Zero test authorship* denotes whether the tool produces its quality verdict without hand-authored quality artefacts beyond the specification (test cases, consumer contracts, or rule configurations); the criterion concerns *authorship*, not input cardinality—most runtime testers consume only the OpenAPI document, but the discriminating question is whether a human must additionally write quality artefacts by hand.

The table reveals a consistent pattern: static tools do not test runtime behaviour, and runtime tools do not assess specification quality. Zero test authorship alone does not discriminate: the generation-based testers (Schemathesis, RESTler, Dredd) already derive test content from the specification automatically, while Pact's value proposition *requires* hand-authored consumer contracts. No existing tool combines static analysis, runtime testing, and specification quality assessment with zero test authorship within a formal methodology designed for GitOps integration; SDT addresses all six dimensions through a single, specification-driven approach. A terminological note: some authors classify the checks of tools like Spectral and Vacuum as *syntactic validation*; we use *static analysis* as the broader umbrella term because these tools also assess documentation substance, style, and convention—concerns exceeding strict syntactic validation while remaining executable without a running API.

Table 2.1: Comparison of API quality approaches across six dimensions.

Tool	Category	Static	Runtime	Spec Quality	Formal Method	GitOps-Native	Zero Test Authorship
Spectral	Linter	✓	✗	∼	✗	✗	—
Vacuum	Linter	✓	✗	∼	✗	✗	—
Redocly	Linter+Docs	✓	✗	∼	✗	✗	—
Optic	Diff/Change	✓	✗	✗	✗	✗	—
Schemathesis	PBT/Fuzzer	✗	✓	✗	✗	✗	✓
RESTler	Stateful Fuzz	✗	✓	✗	✗	✗	✓
Pact	Contract	✗	✓	✗	✗	✗	✗
Dredd	Spec-Driven	✗	✓	✗	✗	✗	✓
Microcks	Mock+Contract	✗	✓	✗	✗	∼	∼
SDT (DriveBy)	**Unified**	✓	✓	✓	✓	✓	✓†

Legend. ✓ = capability present and a primary design goal of the tool; ✗ = capability absent (out of scope for the tool); ∼ = capability partially present, typically as a side effect of the tool's primary purpose rather than a first-class feature; — = the dimension does not apply (e.g. a static linter performs no testing, so test authorship is moot). † Test content—static checks and functional probes—derives from the specification alone; the performance thresholds parameterising P007 are declared in the XSDLC custom resource (Chapter 6), a versioned deployment-policy artefact distinct from the specification. An OpenAPI `x-`extension carrying these thresholds—making the approach single-artefact in the strict sense—is identified as future work in Chapter 11.

2.6.2 The Specification-Driven Gap

We identify three paradigms for API quality assurance, distinguished by what they treat as the primary artefact:

Test-first. Engineers write test cases manually. Tests are the primary artefact; the specification, if it exists, is a secondary reference. The dominant paradigm in practice—hand-authored Postman collections, pytest suites, JUnit tests—is labor-intensive, prone to incompleteness, and disconnected from the specification.

Consumer contract-first. Consumer expectations define the quality contract. Pact and Spring Cloud Contract implement this paradigm. The consumer contract is the primary artefact; the specification is consumed for compatibility verification but not assessed for quality. This paradigm requires active consumer participation and does not address documentation completeness.

Specification-driven. The API specification is the primary artefact. Validation rules, test cases, and quality gates are derived from the specification without manual test authorship. The specification is both the documentation and the testing infrastructure. SDT is the first systematic implementation of this paradigm, grounding it in formal axioms and a principled mapping to validation rules.

The paradigms differ fundamentally in what an engineer must create to enable quality assurance: test cases, consumer contracts, or—in the specification-driven paradigm—only the specification the engineer already maintains for documentation, code generation, and consumer onboarding.

By making the specification the sole input to validation, SDT eliminates the maintenance burden of separate test suites, aligns quality assurance with the artefact that has the strongest organisational incentive for accuracy, and enables full automation within declarative pipelines.

2.6.3 Summary of Research Gaps

Based on the analysis across the preceding sections, our survey identifies seven gaps in the current landscape that SDT and the XSDLC pipeline address:

1. **No unified static and runtime validation.** Existing tools specialize in either static analysis (Spectral, Vacuum, Redocly) or runtime testing (Schemathesis, RESTler, Pact). No tool combines both within a single framework, forcing teams to assemble ad-hoc toolchains with no guarantee of coverage.

2. **No specification quality assessment.** Linters check specification *form*; runtime testers check API *behaviour*; neither assesses whether the specification is *complete* as documentation—the descriptions, examples, error responses, and security definitions consumers need.

3. **No formal axiomatic foundation.** Existing tools implement practical heuristics but cannot articulate *why* a check exists, which axiom it serves, or how checks relate within a principled framework. SDT's three axioms—Completeness, Determinism, and Observability—provide this foundation.

4. **No native GitOps integration.** API validation tools operate as standalone executables or CI plugins; none is designed for reconciliation-based GitOps pipelines where validation results gate declarative promotion.

5. **No specification-derived full quality verdict.** Generation-based testers already derive functional test cases from the specification alone, so single-artefact operation is not itself novel. What no existing tool provides is a *complete* quality verdict—specification quality, functional conformance, and performance—without hand-authored quality artefacts. SDT derives static validation and functional testing wholly from the specification; the performance gate is parameterised by thresholds declared in the XSDLC custom resource, a versioned deployment-policy artefact. The gap is a methodology unifying all three dimensions with zero test authorship while treating the specification's own quality as a first-class, gateable property.

6. **No knowledge transfer assurance.** No widely adopted tool assesses whether the specification is *sufficient for knowledge transfer*—whether a new engineer or an autonomous agent can understand the API contract from the specification alone. SDT's completeness principles (P001–P004) explicitly validate the specification's adequacy as a self-service documentation artefact, reducing cognitive load through codified knowledge [64, 48].

7. **No declarative pipeline provisioning.** Setting up a quality-gated delivery pipeline today requires hours of manual configuration—ArgoCD Applications, webhooks, workflow templates, RBAC, branch protection, Promoter wiring. No existing tool can provision an entire quality gate pipeline from a single declarative specification. The XSDLC custom

resource addresses this gap: one CR (∼35 lines of YAML) generates 45–50 managed Kubernetes objects in under 15 seconds, replacing ClickOps with a version-controlled, reviewable, self-healing specification.

These gaps are not independent: one genuine dependency links them, and the remainder trace back to two shared root causes. The dependency runs from Gap 3 to Gap 1: without a formal methodology stating which quality properties matter, each tool optimises a single property in isolation, producing the observed fragmentation. The first root cause is the field's treatment of the specification as a partial input rather than a complete, self-sufficient contract; it underlies Gaps 2 (testing *from* specifications rather than testing *the* specification), 5 (hand-authored inputs), and 6 (the separation of documentation from quality assurance). The second root cause is the imperative design of existing tools; it underlies Gaps 4 and 7—the assumption that quality gate infrastructure must be assembled manually rather than declared and reconciled, the same assumption that Crossplane and Terraform challenged for cloud infrastructure.

Chapter 3 presents the SDT framework that addresses Gaps 1–6; Chapter 5 presents the DriveBy engine realising Gap 1's unified static-plus-runtime validation in a single tool; and Chapter 6 presents the XSDLC custom resource that addresses Gap 7.

One further thread deserves a forward pointer: the methodology used to *construct* DriveBy itself—iterative, specification-driven development with AI-assisted code generation—is both an empirical validation of SDT's applicability and a meta-analysis of agentic development patterns. Chapter 9 examines how the SDT validation report served as feedback infrastructure for autonomous agents throughout the project's implementation, a claim tested empirically in Section 8.6.

Chapter 3

A Conceptual Framework: Specification-Driven Testing

This chapter presents Specification-Driven Testing (SDT) as a conceptual framework for automated API quality assurance. We define the three axioms that constitute its theoretical foundation, present nine validation principles that operationalize the framework jointly under all three axioms, and specify four validation modes that govern how principles are applied in practice. The chapter defines *what* SDT validates and *why*; Chapter 5 describes *how* the DriveBy framework implements these definitions.

We claim SDT as a *conceptual framework*—a strong theoretical core for a future formal methodology—rather than as a complete software-testing methodology in its own right. A complete formal methodology would additionally cover three elements that are deliberately out of scope here: a formal procedure for deriving tests from the specification (beyond the principle-driven check derivation we present), execution and failure semantics for runtime validation (beyond the binary pass/fail/severity model used by the validation gate), and lifecycle governance over how the principles themselves evolve as the underlying API specification standard changes. These are scoped as future work in Chapter 11.

3.1 The SDT Paradigm

SDT formalises the third of the paradigms identified in Chapter 2 (Section 2.6.2): where test-first requires engineers to author test cases and contract-first requires consumers to publish expectations, SDT derives all validation from the API specification itself—simultaneously the documentation, the contract, and the test oracle.

This is not a tooling distinction. It is a methodological one. SDT treats the specification as a *complete* description of the API contract—not a partial sketch to be supplemented by external tests. *What is being validated*: SDT operates on *two* artefacts that must not be conflated. The static principles (P001–P005, P008, P009) validate the *specification document itself*—they answer "is this OpenAPI document complete, well-structured, and informative enough to support downstream automation?" The runtime principles (P006, P007) validate the *implementation against the specification*—they answer "does the running API behave as the document declares?" One scoping note: the *content* of validation—which checks run, which probes are derived—comes from the specification alone, but the *thresholds* parameterising the performance principle (P007: concurrent users, latency percentiles, success rates) are declared in the XSDLC custom resource (Chapter 6), a versioned deployment-policy artefact—SDT claims specification-sole derivation for test content, not for operational policy. Throughout this thesis, "complete" refers to the specification document unless explicitly qualified otherwise. The three axioms, defined in Section 3.2, apply uniformly to both readings.

These three properties—completeness, determinism, and observability—are axioms: foundational assumptions from which the entire framework follows. We organise SDT as a two-layer structure: the *axiomatic layer* defines three axioms that any SDT-compliant validation must satisfy; the *principle layer* defines nine concrete validation principles. Every principle is bound to satisfy all three axioms simultaneously—it must check a specific facet of contract completeness, produce a deterministic verdict (Section 3.3.1), and emit a structured, machine-readable result. The axioms are therefore framework-wide invariants rather than a partition of the principles; each principle nonetheless has a *governing* axiom that primarily motivates it, alongside the axioms it impacts—a two-sided assignment made explicit in Table 3.1. The separation lets the axioms remain stable as principles evolve and gives a principled admission test for new checks: any proposed check must satisfy all three axioms.

3.2 The Three Axioms

The three axioms of SDT define the minimal conditions under which specification-driven validation is sound. Each axiom addresses a distinct concern: the sufficiency of the specification as input, the reproducibility of the validation process, and the utility of the validation output.

A natural question is whether a fourth axiom—*precision* or *accuracy*, capturing whether the specification's claims correctly reflect the underlying

API—should be added. We chose to fold the concern into Completeness: a specification cannot be fully "complete" in the SDT sense if its content is inaccurate, since inaccurate content fails the dual reading of completeness as both a validation precondition and as transferable knowledge. We acknowledge that an explicit Accuracy axiom would sharpen the framework's vocabulary for specifications that are syntactically complete but semantically incorrect, and record it as a candidate refinement for future work (Chapter 11).

3.2.1 Axiom of Completeness

The API specification must fully describe the API contract; SDT validation must produce a finding for any aspect of the contract that the specification fails to cover.

Completeness requires that the specification contains sufficient information to derive validation rules without external input—not every implementation detail, but everything a consumer needs to understand, integrate with, and depend on the API: endpoints, parameters, request and response schemas, error responses, security mechanisms, and usage examples. Per the folding decision above (Section 3.2), completeness also subsumes *accuracy*: a specification that misrepresents the API is not complete in the SDT sense, however exhaustive its coverage. The static principles measure the descriptive facets on the document itself; the runtime principle P006 (Section 3.3.7) measures accuracy directly against the running implementation.

Completeness is the enabling axiom. Without it, specification-driven validation degenerates into partial checking—the validator can only assess what the specification happens to include, with no way to determine what is missing. Each of the nine principles (Table 3.1) measures one facet of completeness, together giving a multi-dimensional answer to whether the specification fully describes the contract.

Completeness admits a dual reading. In the validation sense, it ensures the specification contains sufficient information to derive validation rules; in the *knowledge transfer* sense, sufficient information for a new engineer—or an autonomous agent—to understand, integrate with, and depend on the API without consulting tribal sources. The two readings coincide: the information needed to validate a contract is precisely the information a consumer needs to use it, so a specification satisfying P001–P004 is by construction an onboarding-complete artefact [48], codified organisational knowledge that reduces cognitive load through self-service interfaces [64] and persists independently of its authors.

A subtle but important consequence: *specification validity is itself a first-class output of SDT, not a pre-condition for applying it.* The very principles

whose Axiom is Completeness (P001–P004 for the static facets, P006 for the runtime check) are what *measure* completeness: the framework does not assume the specification is complete but produces a structured, per-facet report on the degree to which it is, with concrete remediation suggestions. The Axiom describes what SDT *evaluates*, not what it requires as input.

3.2.2 Axiom of Determinism

SDT validation must produce identical results given the same specification, the same API, and the same validation mode.

Determinism requires that the validation process is reproducible: the same inputs must yield the same outputs regardless of execution environment, time, or operator identity, eliminating subjectivity. This is critical for automation—a quality gate that produces different results on repeated execution cannot serve as a reliable gate. Determinism is satisfied at the level of the whole framework by the parallel-application semantics of Section 3.3.1: each principle is atomic and side-effect-free, the aggregation function is canonical, and the report is the same under any execution order or parallelism choice. It is therefore a structural property of how the principles compose, not a property of any one principle.

Context-dependent APIs and the determinism axiom. Do context-aware APIs—those that legitimately return different responses for the same request based on tenant, locale, time-of-day, or feature-flag state—break the Determinism axiom? They do not: the axiom constrains *validation results*, not API responses. A P006 functional test derived from a specification asserts the properties the specification declares (status code, schema shape, declared header presence), not the literal response body, so a multi-tenant API returning tenant-specific data satisfies P006 deterministically as long as every response conforms to the declared schema. Asserting *literal content* would be stronger, but for context-dependent APIs it would shift the determinism burden onto the specification's examples, which would then have to be declared per tenant, locale, or flag state; SDT therefore scopes P006 assertions to contract conformance (Section 3.3.7). What *does* break the axiom is non-determinism in the validation logic itself (random sampling, time-sensitive thresholds, retries with hidden side effects), which SDT explicitly forbids: such tests are not eliminated but parameterised to assert only specification-declared properties. Specifications that declare context-dependent behaviour (e.g. via `x-context` extensions or per-tenant servers) are first-class SDT inputs; Completeness covers these declarations too: such behaviour is part of the contract, so a specification omitting it is incomplete in the SDT sense.

Determinism is especially critical when the consumer of validation results

is an autonomous agent: an agent that receives different results on repeated runs cannot distinguish genuine regressions from noise, and therefore cannot be trusted for autonomous corrective action [77].

3.2.3 Axiom of Observability

SDT validation must produce structured, machine-readable, and actionable reports for every principle it evaluates.

Observability requires that validation output is structured, quantifiable, and suitable for automated consumption: results a CI pipeline can gate on, a dashboard can display, and an engineer can act on without interpreting free-text narratives. Where a code review yields unstructured prose, SDT produces structured reports with per-principle pass/fail status, severity classifications, specific check results, and suggested remediations. Observability is a property of every principle's output: each f_{p_i} emits a structured, typed, machine-readable `PrincipleResult`, and aggregation preserves that structure into the final report. Functional-test outcomes (P006) are treated identically—their remediation-relevant information (failing endpoint, payload, status code) appears in the same form as static findings, so downstream automation acts on runtime and static results through one uniform interface.

"Automated consumption" is deliberately broad: it encompasses CI/CD pipelines gating on exit codes, dashboards visualising quality trends, and autonomous engineering agents that parse validation reports and generate remediation code [77, 23]. Because results are structured and deterministic, they also support longitudinal tracking of quality trends across versions and APIs.

Observability admits a second reading that the validation *conclusion*—not merely the *output*—should be grounded in observable evidence. Some principles read this evidence from the specification document (e.g. whether a security scheme is declared, whether an error response is documented), others from the deployed system at runtime (e.g. whether the declared scheme is actually enforced, whether responses conform to the declared schema). The framework places no special weight on one source over the other: both are first-class inputs to validation, and both contribute structured findings under the Observability axiom.

3.3 The Nine Principles

Each SDT principle defines a specific dimension of API quality, traces its rationale to one or more axioms, classifies its severity, and enumerates the

concrete checks it performs.

Severity semantics. **Critical**: failure indicates the specification cannot support meaningful downstream automation (test generation, contract enforcement, security gating); the validation gate blocks promotion (exit code 1). **Warning**: failure indicates a quality gap that degrades the specification's usefulness without preventing automation; the gate surfaces it without blocking (exit code 0). The precise severity-to-gate mapping is defined in Section 3.4.

3.3.1 Validation as Parallel Application and Aggregation

Before describing the principles individually, we fix the formal model that governs how they combine: parallel application over the same input, not function composition.

Let s denote an API specification (an OpenAPI 3.x or Swagger 2.0 document, normalised through the abstraction layer of Section 5.3), and let f_{p_i} denote the principle-specific validation function mapping the specification to a structured `PrincipleResult`. A *validation policy* f_p is determined by a subset of principles $\{p_1, \ldots, p_n\}$ selected by the validation mode (Section 3.5). The policy is realised as

$$f_p(s) \;=\; \mathrm{agg}\big(f_{p_1}(s),\, f_{p_2}(s),\, \ldots,\, f_{p_n}(s)\big),$$

where agg is the deterministic aggregation function that assembles the per-principle results into the structured validation report (Section 5.8). Each principle function receives the *same* specification s as input; no principle consumes the output of another.

This is *not* function composition: there is no chain $f_{p_n} \circ f_{p_{n-1}} \circ \cdots \circ f_{p_1}$. Three consequences follow directly from this choice.

Determinism is preserved by construction. Each f_{p_i} is atomic and side-effect-free: it reads only s and writes only its own `PrincipleResult`, with no shared mutable state, ordering effect, or cross-principle data flow. The aggregation function consumes the set of per-principle outputs $\{f_{p_i}(s)\}_{i=1}^{n}$ under a fixed canonical ordering by principle identifier, so any permutation of the per-principle calls produces the same report—satisfying the Axiom of Determinism by construction rather than by convention.

Validation modes are modular subsets, not modified pipelines. Each validation mode (`minimal`, `strict`, `test-ready`, `test-only`) corresponds to a subset of the principle index set, not to a reordering or transformation of a fixed pipeline (Section 3.5). Adding a new principle registers an additional $f_{p_{n+1}}$ without disturbing the principles that preceded it; restricting a mode drops an index from its subset. Composition, by contrast, would force modes to encode evaluation order and partial-failure truncation semantics.

Principles can be evaluated concurrently without coordination. Because no f_{p_i} reads any other principle's output, the formalism permits parallel, sequential, or distributed evaluation; the current implementation runs them sequentially for operational simplicity (Section 5.6) without affecting $f_p(s)$.

The aggregation function agg introduces no decisions of its own beyond canonical ordering and summary counting; the report structure it produces is described in Section 5.8. This formal commitment is what makes validation a deterministic "sensor" in the control-theoretic framing of Chapter 7 (Section 7.6) and the SDT report a stable feedback signal for autonomous agents (Chapter 9).

Mode coverage. Each principle entry below carries a **Mode behaviour** note naming the static modes (Section 3.5) in which it runs: *Mode-uniform* when the check set is identical in every mode in which the principle runs, *Mode-specific* (with the exact per-mode subset) when it differs, plus an explicit statement when a mode skips the principle. The mode–check matrix is thus explicit at the principle level, not only in Table 3.2.

3.3.2 P001: OpenAPI Specification Compliance

Severity: Critical. **Status:** Implemented. **Mode behaviour:** Mode-uniform—P001 applies its identical seven-check set in every static mode (minimal, test-ready, strict).

P001 validates that the API specification conforms to the OpenAPI standard [51] and is structurally sound. A specification that fails P001 cannot be reliably consumed by code generators, documentation renderers, or downstream validators—nor meaningfully assessed by any other principle; P001 is therefore a prerequisite for all other principles and the one principle present in every mode.

P001 performs seven checks: (1) the specification version is present and valid; (2) required info fields (title, version) exist; (3) at least one path is defined; (4) components are structurally valid; (5) the overall specification structure passes validation via the `APISpec.ValidateStructure()` adapter method, which delegates to the underlying parser's schema validator; (6) no duplicate `operationId` values exist across operations; and (7) all HTTP methods used are standard (GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS, TRACE). For OpenAPI 3.x specifications, the structural check (5) additionally verifies null type semantics: OpenAPI 3.1.0 permits `type: "null"` directly, whereas OpenAPI 3.0.x requires the nullable keyword—a specification that uses the wrong null representation for its declared version fails the structural check.

3.3.3 P002: API Documentation Quality

Severity: Critical. **Status:** Implemented. **Mode behaviour:** Mode-specific—*strict* mode performs all ten checks below; *test-ready* mode performs the test-critical subset (operation summaries, descriptions, and `operationId`s; parameter descriptions; request/response examples; schema and enum descriptions), skipping the contact and license checks; not evaluated in minimal mode.

Severity escalation note. P002 was originally classified as **Warning** on the rationale that incomplete documentation degrades a specification's usefulness without strictly preventing automation. Operational evidence from the controlled and large-scale evaluations (Chapter 8) reversed this judgement: specifications missing operation summaries, parameter descriptions, schema descriptions, or response examples systematically prevent downstream agents and code generators from producing usable artefacts, and missing examples cascade into the runtime principles—without them, the functional and performance pipelines (P006, P007) can only synthesize input data, reintroducing exactly the non-determinism the framework forbids. The principle was escalated to **Critical** in the source code (commit `c96d9c7`, "Make P002/P003/P004 critical"); the severity stated here reflects those source-of-truth constants, and the same escalation applies to P003 and P004 below for parallel reasons. The escalation has a load-bearing empirical consequence: 84% of public APIs in the APIs.guru sample now block at the static layer rather than degrade gracefully (Chapter 8)—the quantitative form of the systemic specification-quality gap that motivates the framework. Strict mode is therefore a true quality gate rather than an advisory layer; teams needing a non-blocking baseline use minimal or test-ready mode (Section 3.5).

P002 assesses whether the specification serves as effective documentation: a specification can be structurally valid (passing P001) yet practically useless, with endpoints lacking descriptions and schemas lacking examples.

P002 performs ten checks across three scopes. At the *API level*: (1) the API has a general description; (2) contact information is provided; (3) license information is specified. At the *operation level*: (4) all operations have summaries; (5) all operations have detailed descriptions; (6) all operations have unique `operationId` values; (7) all parameters have descriptions; (8) all request and response bodies include examples. At the *schema level*: (9) all component schemas have descriptions; (10) all enum values are documented within their schema descriptions.

P002 addresses a gap identified in Chapter 2: existing linters verify specification *form*, whereas P002 systematically verifies documentation *substance*—that every element a consumer interacts with is documented.

3.3.4 P003: Error Handling Standards

Severity: Critical. **Status:** Implemented. **Mode behaviour:** Mode-specific—*strict* mode performs all seven checks below; *test-ready* mode performs exactly two: check (1), 4xx error responses documented, and check (5), error schemas contain structured fields (`message`, `code`, or `details`), skipping the 5xx, reusable-component, and format-consistency checks. P003 is not evaluated in minimal mode.

P003 validates that the specification documents error responses comprehensively and consistently. Undocumented error responses are a primary source of integration failures: consumers must reverse-engineer unmentioned 403 or 422 responses from raw HTTP traffic.

In strict mode, P003 performs seven comprehensive checks: (1) all operations document at least one 4xx error response; (2) all operations document at least one 5xx error response; (3) error responses include descriptive error codes (via non-empty descriptions); (4) error responses include error messages; (5) error response schemas contain structured fields (`message`, `code`, or `details` properties); (6) common error responses (400, 401, 403, 404, 500) are defined as reusable components in `components.responses`; and (7) error response formats are consistent across operations (all errors share the same schema property structure).

The consistency check (7) is particularly valuable: APIs that return `{“message”: ...}` for 400 errors but `{“error”: ..., “code”: ...}` for 500 errors impose unnecessary complexity on consumers, so the implementation flags any specification where more than one distinct error-schema property structure appears.

Uniformity vs completeness. Check (7) raises a sharper conceptual question: is the consistency it enforces a property of *completeness* (the specification's coverage of error formats is incomplete because it omits a uniform-format declaration), or a distinct property of *uniformity* that should arguably be a fourth axiom? The framework currently folds it into completeness on the grounds that a multi-format specification is missing the meta-information "which format applies to which class of error"—i.e. the heterogeneity is a completeness gap at the meta-level. We acknowledge that an explicit *Axiom of Uniformity*, governing format consistency across operations, schemas, and error responses, would sharpen the framework's vocabulary; the question of whether to introduce it is recorded as a candidate refinement in Chapter 11 alongside the precision/accuracy axiom.

3.3.5 P004: Schema Definitions

Severity: Critical. **Status:** Implemented. **Mode behaviour:** Mode-specific—*test-ready* mode checks schema existence plus type specifications (every schema and property must declare a type) without enforcing constraints, so that test-data generators can produce typed values; *strict* mode performs all ten checks below. P004 is not evaluated in minimal mode.

P004 validates that request and response schemas are precise, constrained, and complete—not merely present but informative. A schema declaring a field as type `string` without length, pattern, or format constraints provides the consumer with no useful validation guidance.

In strict mode, P004 performs ten comprehensive checks: (1) all path parameters have schemas; (2) all query parameters have schemas; (3) all header parameters have schemas; (4) all request bodies have content schemas; (5) all schemas specify data types; (6) all schemas have appropriate constraints; (7) all required fields listed in `required` arrays exist in `properties`; (8) all enum values are non-null; (9) all numeric fields have range constraints (`minimum`, `maximum`); and (10) all string fields have length constraints (`minLength`, `maxLength`) or pattern restrictions.

All checks are applied recursively through the schema tree, including nested object properties and array items—a top-level object may be well-defined while a nested field is typed as a bare `object` with no properties. This aligns with the JSON Schema specification's [41] principle that schemas should be as restrictive as the domain permits.

3.3.6 P005: Security Standards

Severity: Critical. **Status:** Implemented. **Mode behaviour:** P005 is evaluated only in *strict* mode, where it performs all seven checks below. Its exclusion from minimal and test-ready is a deliberate design decision: test-readiness asks whether meaningful test data can be derived from the specification, and authentication is exercised at the runtime layer rather than as a precondition for test generation.

P005 validates that the specification defines security mechanisms comprehensively and consistently. Security misconfigurations in API specifications—missing authentication requirements, undefined schemes, inconsistent enforcement—are among the top risks identified by the OWASP API Security project [53], and their omission leaves consumers guessing and automated tools unable to verify compliance [13].

P005 performs seven checks: (1) at least one security scheme is defined in `components.securitySchemes`; (2) global security requirements are set at

the top level; (3) all operations are covered by either global or operation-level security (operations inherit global security when no operation-level security is declared); (4) OAuth2 and OpenID Connect schemes are referenced with scopes in at least one security requirement; (5) API key schemes include descriptions; (6) API key schemes specify the `name` and `in` fields (header, query, or cookie location); and (7) all security scheme names referenced in security requirements are defined in components.

Check (7) addresses a subtle but dangerous inconsistency: a security requirement referencing a scheme name absent from `components.securitySchemes` appears to enforce security whose enforcement is undefined; P005 scans both global and per-operation requirements for such phantom references.

Intentionally public APIs. P005 applies even to a deliberately public API, but the obligation it imposes is *declaration*, not protection. OpenAPI distinguishes an operation that explicitly declares it requires no authentication (an empty security requirement, `security: []`) from one whose specification is simply silent: the former documents a design decision that consumers and validators can rely on; the latter leaves authentication undefined. P005 findings on a public API are therefore resolved by declaring the public posture explicitly, not by adding authentication the API does not need.

3.3.7 P006: Functional Testing

Severity: Critical. **Status:** Defined theoretically; implementation as a `PrincipleChecker` wrapper is pending. **Mode behaviour:** Runtime principle—exercised by *test-only* mode through the functional-testing pipeline (and the `functional-test` gate check type), not by any static mode.

P006 validates that the API's runtime behaviour matches the specification; unlike the static principles, it requires a live API instance. It derives test cases from the specification—HTTP requests with appropriate parameters, headers, and bodies—and verifies that responses conform to the declared schemas, status codes, and content types.

The DriveBy framework already implements the underlying functional testing logic (Chapter 5); the pending work is integrating it into the unified principle evaluation pipeline so that it contributes to the structured validation report.

P006 operationalizes the Axiom of Determinism: given the same specification and the same API, functional tests must produce the same pass/fail results. This requires that test inputs are derived deterministically from the specification rather than generated randomly (distinguishing SDT's functional testing from property-based approaches like Schemathesis [28]). Determin-

istic derivation relies in practice on declared example data: parameter and request-body examples let the test generator produce the same typed, valid inputs on every run—precisely the precondition that P009 (Section 3.3.10) verifies before runtime testing begins. Two operational consequences follow. First, example values flow into test requests and validation reports, so authors must not embed real or sensitive data in examples; the framework assumes this synthetic-data discipline rather than enforcing it. Second, authentication credentials are deliberately *not* derived from example data: they are supplied as operational configuration (Chapter 7), keeping secrets out of both the specification and the structured report.

Assertion scope. P006 asserts *contract conformance*—status codes, response schema shape, declared types and headers—by design. Semantic correctness of response *content* (whether a returned value is the *right* value, not merely well-formed) would require an oracle the specification cannot supply; it is out of P006's scope and recorded as a limitation in Chapter 11, with example-derived content assertions scoped as a natural future extension.

3.3.8 P007: Performance Testing

Severity: Critical. **Status:** Defined theoretically; implementation as a `PrincipleChecker` wrapper is pending; the load-test runner that realises it is operational. **Mode behaviour:** Runtime principle—exercised by *test-only* mode through the load-testing pipeline (and the `load-test` gate check type), not by any static mode.

P007 validates that the API meets performance expectations by measuring response times, throughput, and error rates under load; like P006, it requires a live API instance. It operationalizes the Axiom of Observability: it produces latency distributions, throughput measurements, and error rate statistics that can be compared against configurable thresholds and tracked longitudinally across API versions.

Performance measurements are environmentally sensitive: the same API under the same load profile yields different raw latencies across runs, so per-sample threshold checks could flip between pass and fail, violating the Determinism axiom in spirit. P007 therefore evaluates its thresholds only over statistically robust aggregates of a sustained run—latency percentiles such as P95 and success-rate ratios—never individual samples. Provisioning a representative environment remains the operator's responsibility: the framework makes measurement and verdict derivation deterministic, but cannot make under-provisioned or noisy infrastructure behave deterministically.

The pending work mirrors P006: integrating the existing performance testing logic into the unified principle evaluation pipeline.

3.3.9 P008: API Versioning Strategy

Severity: Warning. **Status:** Implemented. **Mode behaviour:** P008 is evaluated only in *strict* mode, performing all seven checks below; minimal and test-ready skip it (versioning discipline does not affect whether meaningful tests can be generated).

P008 validates that the API follows a disciplined versioning strategy. APIs evolve; without explicit versioning, deprecation notices, and migration guidance, consumers face silent breaking changes. P008 ensures the specification communicates its version lifecycle clearly, following the Semantic Versioning convention [58].

P008 performs seven checks: (1) the API version is specified in the info section; (2) the version follows Semantic Versioning format (MAJOR.MINOR.PATCH), validated via the regular expression `^\d+\.\d+\.\d+$`; (3) the versioning strategy is documented in the API description (detected by keyword match for "version"); (4) deprecated operations include deprecation details in their descriptions (operations marked `deprecated: true` must contain the word "deprecat" in their description); (5) breaking changes are documented or a changelog is referenced; (6) version compatibility (backward compatibility) is addressed; and (7) migration or upgrade guides are referenced.

Checks (3)–(7) are distinctive: where most tools verify only that a version string exists and follows semver, P008 requires that the specification communicates its *versioning practices*—transitions, changes, and migration guidance—by scanning `info.description` for relevant keywords, transforming the version field from a static label into a living contract about the API's evolution.

3.3.10 P009: Test Readiness

Severity: Warning. **Status:** Implemented. **Mode behaviour:** Mode-uniform—P009 runs in *test-ready* mode (its four checks are the core of that mode's verdict) and performs the identical check set whenever it is evaluated.

P009 validates that a specification provides sufficient detail for meaningful functional and load testing: without adequate examples, typed schemas, and response definitions, automated test generators fall back to generic placeholder values that real API validation logic rejects, producing false failures.

P009 performs four checks:

1. *Path parameters have examples or typed schemas.* Without an example or a typed schema, parameter substitution falls back to untyped strings

(`"test-value"`) that fail server-side validation for integer, UUID, or enum parameters.

2. *Request bodies have examples or typed schema properties.* The functional tester's request body builder prioritises explicit examples, then falls back to schema-based generation. The fallback is itself a determinism risk: values must be synthesized purely from the schema (fixed representative values per declared type), never sampled randomly—one reason P009 flags missing examples, steering specifications toward trivially deterministic test derivation. If neither is available, generated payloads are rejected by any API with input validation.

3. *2xx responses have schemas defined.* Without response schemas, functional tests degrade to status-code-only checks with no response body validation.

4. *2xx responses have examples defined.* Advisory *within* P009: a miss is surfaced as a finding but does not by itself fail the principle—a status distinct from P009's principle-level Warning severity, which governs the gate mapping (Section 3.4). Response examples improve test reporting but are not required for the contract-conformance testing P006 performs; they would become necessary only for content-level assertions, which are outside P006's scope (Section 3.3.7).

P009 thus secures the preconditions for deterministic test execution: generated test data is typed and meaningful rather than dependent on arbitrary fallback heuristics.

Why a principle and not a mode. Is P009 redundant—with the test-ready *mode*, or with P001 plus partial P002–P004? No, for three reasons. First, modes and principles are different categories: a mode is a *selection* of principles, a principle a measurable property with its own verdict. Test-ready mode is the set {P001, P002, P003, P004 (reduced check sets), P009}; without P009 as a principle, the mode's defining question—can meaningful test data be derived from this specification?—would have no dedicated, reportable verdict. Second, P009's checks are *disjunctive preconditions* that no other principle expresses: P002 asks the documentation question "do all request and response bodies have examples?", P009 the strictly weaker test-derivability question "does every parameter and body have an example *or* a typed schema?"—the minimal condition for typed, valid test data instead of untyped fallbacks. A specification with no examples but fully typed schemas *fails* P002 yet *passes* P009—precisely the case in which functional testing is still meaningful—and the reduced P002–P004 check sets, being conjunctive, cannot express

this disjunction. Third, keeping P009 a principle preserves composability: any future mode or gate can include the test-derivability verdict without re-deriving it from fragments of other principles.

3.4 Severity, Status, and the Gate Decision

The nine principles are catalogued in Table 3.1. The axiom columns encode a deliberately two-sided claim: the *governing axiom* is the one that primarily motivates the principle, while the *impacted axioms* record that *every principle simultaneously contributes to all three*—it measures one facet of completeness, produces a deterministic verdict under the parallel-application semantics (Section 3.3.1), and emits structured findings satisfying Observability. The governing assignment therefore groups, rather than partitions, the principles: the completeness-governed group (P001–P004) enforces descriptive coverage, the observability-governed group (P005, P007, P008) surfaces measurable signals into the structured report, and the determinism-governed group (P006, P009) guarantees repeatable test outcomes—P006 by deriving test cases deterministically from the specification, P009 by securing the preconditions under which that derivation produces typed, stable inputs.

Table 3.1: Catalogue of the nine SDT principles. **Governing** = the axiom that primarily motivates the principle; **Impacted** = the remaining axioms, which every principle also satisfies (**C**ompleteness, **D**eterminism, **O**bservability). Severity reflects the source-of-truth constants in `driveby-cli/internal/types/report.go`.

ID	Principle	Governing Axiom	Impacted Axioms	Severity	Status
P001	OpenAPI Specification Compliance	Completeness	D, O	Critical	Implemented
P002	API Documentation Quality	Completeness	D, O	Critical	Implemented
P003	Error Handling Standards	Completeness	D, O	Critical	Implemented
P004	Schema Definitions	Completeness	D, O	Critical	Implemented
P005	Security Standards	Observability	C, D	Critical	Implemented
P006	Functional Testing	Determinism	C, O	Critical	Planned
P007	Performance Testing	Observability	C, D	Critical	Planned
P008	API Versioning Strategy	Observability	C, D	Warning	Implemented
P009	Test Readiness	Determinism	C, O	Warning	Implemented

The severity classification—critical versus warning—governs how validation results translate to quality gate decisions through a deterministic exit code mechanism: if any principle with `severity: critical` has `Passed: false`, the overall report status is `"failed"` and the process exits with code 1; if only warning-severity principles fail, the status remains `"passed"` with exit

code 0. This binary mapping is the mechanism by which SDT integrates with CI/CD quality gates.

The severity assignment reflects a core SDT design principle: *static validation gates runtime testing*—functional (P006) and performance (P007) alike, since load profiles, like functional probes, are derived from the documented operations and payloads. The completeness-of-contract principles (P001–P005) are all critical because an incomplete specification cannot produce meaningful test results from either runtime principle: without examples and descriptions (P002) generated request data is rejected as false failures, without documented error schemas (P003) negative assertions cannot verify error shapes, without typed schemas (P004) generators fall back on placeholders that fail server-side validation, and misconfigured authentication (P005) blocks all endpoint access. Only versioning (P008) and test readiness advisory checks (P009) remain at warning severity—quality dimensions that degrade gracefully rather than preventing test execution entirely.

3.5 Validation Modes

SDT defines four validation modes that control which principles are evaluated, enabling incremental adoption: a team can begin with minimal validation and progressively enable stricter checks as its specification matures. The static mode ladder encodes increasing validation *cost* as much as strictness, so mode selection is simultaneously a strictness and a budget choice; test-only sits outside the ladder, trading static checks for runtime ones. Table 3.2 presents the mode-to-principle mapping.

Minimal mode evaluates only P001 (OpenAPI Specification Compliance): a fast baseline gate for pull request pipelines where rapid feedback is prioritised. A specification that passes is structurally valid and safe for code generation, but its documentation quality, error handling, and security definitions are unchecked.

Strict mode evaluates P001 through P005 and P008—all implemented static principles—providing comprehensive specification quality assessment without a live API instance. One might expect strict mode to include every principle except P009; the boundary is deliberate: strict mode is the portion of a pre-merge gate that runs on the specification alone, and a full gate adds the dynamic test-only stage once a deployed environment is available. Strict mode suits release pipelines, the static half of pre-merge gates, and periodic audits.

Test-only mode skips all static validation: the principle registry returns no checkers, and the runtime testing pipeline—functional endpoint testing

Table 3.2: Validation modes and the principles each mode evaluates. ✓ = full check set; ∼ = partial check set (detailed in the notes below); ✓° = runtime principle exercised outside the static checker registry. The registry's `ForMode()` method returns only the checkers listed for each mode.

Mode	**P001**	**P002**	**P003**	**P004**	**P005**	**P006**	**P007**	**P008**	**P009**
Minimal	✓								
Test-ready	✓	∼‡	∼§	∼†					✓
Strict	✓	✓	✓	✓	✓			✓	
Test-only						✓°	✓°		

° P006 and P007 are defined theoretically but not yet implemented as `PrincipleChecker` wrappers. In the current implementation, `ForMode(test-only)` returns no static checkers; the mode instead triggers the runtime testing pipeline (functional and load tests) directly via separate CLI commands.

† P004 in test-ready mode checks schema existence and type specifications only, without constraint validation.

‡ P002 in test-ready mode checks examples, schema descriptions, enum descriptions, and operation documentation, but skips contact and license information.

§ P003 in test-ready mode checks that operations document 4xx error responses and that error schemas include structured fields (message, code, details), but skips 5xx requirements, component-level reuse, and format consistency.

and load testing—runs directly against a live API instance (operational via the `function-only` and `load-only` CLI commands, pending the P006/P007 wrappers noted in Table 3.2). It serves scenarios where the specification has already been validated statically, decoupling static and runtime validation into separate pipeline stages.

Test-ready mode evaluates P001, P009, and the reduced check sets of P002–P004 (Table 3.2 footnotes). It is a fast pre-flight check answering: *is this specification good enough to produce meaningful test results?* Without it, functional tests send placeholder values that are rejected by API validation, and load tests run for minutes before revealing that the specification was incomplete; test-ready mode runs in seconds and gates whether testing is worth executing.

The four modes map onto the standard cost-increasing sequence of organisational quality gates: *minimal* as a fast gate on every commit, *test-ready* as a pre-flight gate before functional testing pipelines, and *strict* as the static half of a pre-merge gate. *Test-only* maps onto a pre-merge, post-merge, or release gate depending on environment availability: functional tests need a running instance of the changed API (pre-merge where ephemeral preview environments are cheap, post-merge or release where only shared staging exists), while load tests, the most expensive stage, typically gate release promotion.

3.6 Summary

This chapter defined SDT as a two-layer conceptual framework: three axioms—Completeness, Determinism, and Observability—and nine validation principles derived from them, applied under four validation modes that enable incremental adoption. What is presented here is the theoretical core for a future formal methodology, whose remaining elements are scoped as future work in Chapter 11. Chapter 5 describes how the DriveBy framework implements these definitions.

Chapter 4

System Architecture

Chapter 3 defined Specification-Driven Testing as three axioms and nine validation principles. Before any single component is examined, this chapter establishes the architecture of the *system* that realises the framework: its components, its boundary, its infrastructure, and how the work of turning a specification into a promotion decision is divided.

The system is larger than the tool: the DriveBy CLI is only the validation engine and cannot, by itself, gate a delivery pipeline. The system under study comprises four components—the DriveBy CLI, which evaluates specifications against the SDT principles; the XSDLC Crossplane composition, which generates per-application pipeline infrastructure; the generated gate infrastructure itself (event sources, sensors, validation workflows, ingress routes); and the GitOps Promoter, which converts validation verdicts into promotion decisions. Removing any one of them breaks the chain from specification to promotion.

Equally important is what the system is *not*. Kubernetes and Crossplane are not components but the system's *infrastructure substrate*: the control plane that hosts it, reconciles its declared resources, and provides the shared platform services (ArgoCD, the Argo Events and Argo Workflows engines, ingress, certificate management) that the system references but does not install. This system-on-substrate separation recurs throughout the thesis; the operational evaluation (Chapter 8) demonstrates that the system can be installed on, coexist with, and be surgically removed from a pre-existing substrate.

4.1 System Context

Figure 4.1 fixes the system boundary: the dashed boundary encloses the four components, with the substrate band—the Kubernetes and Crossplane

control plane plus inherited platform services—underneath. Outside remain only the entities the system does not control: the API developer, the platform operator, the target API and its OpenAPI specification, the external GitHub repositories (Chapter 7), and the structured validation report—the system's primary output.

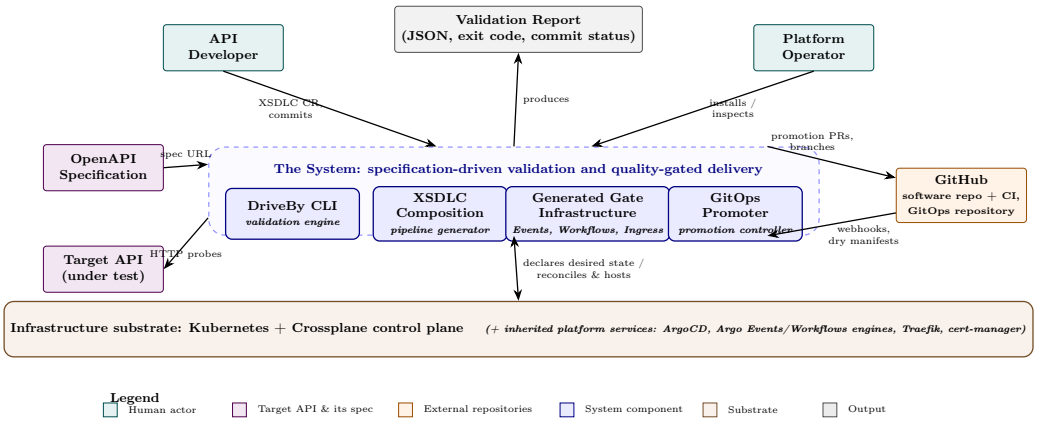

Figure 4.1: System context. The dashed boundary encloses the system under study: the DriveBy CLI (validation engine), the XSDLC composition (pipeline generator), the generated gate infrastructure, and the GitOps Promoter (promotion controller), resting on the infrastructure substrate that hosts and reconciles it. External entities: human actors, the target API and its specification, the GitHub repositories, and the validation report.

Two boundary decisions deserve justification. First, the *GitOps Promoter is inside the boundary*: it consumes the gate verdicts (as commit statuses) and performs the promotion the gates exist to control; placing it outside would sever the defining feedback loop—validation results driving promotion decisions—and reduce the system to a passive report generator. Second, *ArgoCD and the Argo engines are substrate, not system*: they pre-exist on the target cluster, serve many tenants, and are referenced rather than installed by XSDLC. The system's contribution is the declarative configuration that instructs these engines, not the engines themselves.

4.2 The Three-Layer Model

The system divides its work into three abstraction layers, each described by one of the implementation chapters that follow. A standalone validation CLI, however well-designed, cannot satisfy the operational demands of continuous delivery at scale; three requirements drive the layering.

First, *execution must be automated*. A manually invoked CLI is useful for debugging but cannot serve as a quality gate, which must fire on every relevant event—a promotion pull request, a scheduled regression, a new deployment—without human intervention.

Second, *configuration must be declarative*. Imperative validation ("run this command with these flags") lives in scripts, CI pipelines, or developer knowledge; declarative validation ("this API must pass strict SDT validation before promotion to staging") is a versioned, reconcilable Kubernetes resource that supports drift detection, audit trails, and self-healing.

Third, *results must drive automated decisions*. The CLI's JSON report and exit code suffice for a human reviewer but not for a GitOps reconciliation loop, which requires structured commit statuses, webhook callbacks, and status conditions rather than stdout parsing.

These requirements map directly to the three layers, summarised in Table 4.1: the CLI (Chapter 5) provides the *validation logic*, the Kubernetes infrastructure (Chapter 6) the *execution and provisioning layer*, and the GitOps pipeline (Chapter 7) the *decision layer*.

Table 4.1: The three abstraction layers of the system.

Aspect	**CLI (Ch. 5)**	**Kubernetes (Ch. 6)**	**GitOps (Ch. 7)**
Execution	Manual invocation	Event-driven workflows	Promotion-triggered
Configuration	CLI flags, env vars	XRD spec fields, Helm values	Git-versioned manifests
Output	JSON to stdout	Workflow artefacts	Commit statuses
State	Transient (process)	Pod lifecycle	Reconciled desired state
Promotion	Human decision	Sensor-triggered	Automated (commit status)

4.3 From CLI Tool to Custom Resource

The three-layer structure is the end point of an architectural evolution through five stages, each subsuming the previous while adding a new capability. Tracing them explains *why* the system has the shape that Figure 4.1 fixes.

4.3.1 Stage 1: Shell Script

The earliest form of API validation is a shell pipeline: `curl` fetches the specification, `jq` extracts fields, and conditional logic checks for missing descriptions or undocumented error codes. It requires nothing beyond a POSIX shell, but lacks structure (ad-hoc string matches), reusability (a

custom script per API), and determinism (each developer encodes different checks and thresholds), and cannot produce structured reports, compose into pipelines, or participate in automated decision-making.

4.3.2 Stage 2: Go CLI

The DriveBy CLI (Chapter 5) addresses these limitations with typed principles, a version-agnostic spec abstraction (`APISpec`), a deterministic validation engine, and structured JSON reporting. Of the nine SDT principles (P001–P009), the CLI implements seven as `PrincipleChecker` instances (P001–P005, P008, P009); P006 (functional testing) and P007 (performance testing) are handled by dedicated test runners (Section 6.3.2). The same principle set runs on every specification, producing comparable, machine-readable results from a self-contained binary with zero runtime dependencies.

4.3.3 Stage 3: Container Image

Packaging the CLI as an OCI container image (`ghcr.io/meter-peter/driveby`) adds portability: the same binary runs on a developer's laptop or inside any container runtime—Docker, containerd, CRI-O—without dependency management. The image is the unit of deployment for all subsequent stages; workflow steps, CI jobs, and Crossplane-managed workflows all reference it.

4.3.4 Stage 4: Argo Workflow Step

Embedding the container image as a step in an Argo Workflows DAG adds event-driven execution, dependency ordering, artefact persistence, and exit handling. Argo Events triggers the workflow from GitHub webhooks; the workflow runs validation alongside health checking, commit-status posting, and PR commenting, and posts results back to the Git control plane—transforming the CLI from a manual tool into an automated quality gate.

4.3.5 Stage 5: Crossplane Custom Resource

The final stage—the `XSDLC` custom resource described in Section 6.3—declares the entire pipeline as a single Kubernetes resource: from one CR ($\sim$35 lines of YAML) declaring the desired end state, Crossplane's reconciliation loop provisions the complete gate infrastructure.

The key insight is that each stage *subsumes* the previous: the CR still runs the same CLI binary, inside the same container image, as a step in the same Argo Workflows DAG. Nothing is discarded; each layer adds a

capability (portability, automation, declarative management) while preserving the core validation logic. The critical enabler is the CLI's own architecture (Section 5.1.1), which makes the validation logic portable across execution environments—the same principle checkers run in a developer's terminal and inside an Argo Workflow container. The CLI was not retrofitted for Kubernetes integration; it was designed from the outset to support multiple deployment models.

4.4 Chapter Roadmap

The three chapters that follow descend through the layers: Chapter 5 opens the DriveBy binary, Chapter 6 describes the execution and provisioning layer, and Chapter 7 the decision layer.

Chapter 5

The DriveBy CLI: Internal Architecture

Chapter 4 situated the DriveBy CLI within the overall architecture as the validation-logic layer; this chapter descends into that component and describes how it implements the SDT definitions of Chapter 3. The presentation follows the data flow through the tool—specification loading, normalisation, principle evaluation, report generation, CLI output—and each section explains the design decisions behind a subsystem and references the source code that realises them.

The scope of this chapter is the DriveBy binary itself—its internal packages, interfaces, and data structures; the Kubernetes infrastructure that hosts DriveBy as a quality gate is presented in Chapter 6, and the GitOps pipelines that trigger and consume its output in Chapter 7.

Where Figure 4.1 (Chapter 4) drew the boundary around the whole delivery system, Figures 5.2 and 5.3 open the DriveBy binary to expose its internal packages and central interfaces.

5.1 Architecture Overview

5.1.1 Design Goals

The DriveBy architecture is shaped by five design goals derived from the SDT axioms:

1. **Version agnosticism.** Principle checkers must operate identically on OpenAPI 3.0.x, OpenAPI 3.1.0, and Swagger 2.0 specifications. No checker may contain version-specific branching logic.

2. **Extensibility.** Adding a new validation principle must not require modifying existing checkers or the engine. The architecture must support open extension without closed modification.

3. **Determinism by construction.** Static validation must be free of side effects, randomness, and environment-dependent behaviour, producing identical reports for identical inputs. The same discipline extends to dynamic validation: runtime test cases are derived deterministically from the specification, never randomly (Section 5.7).

4. **Machine-readable output.** All validation output must be structured JSON suitable for consumption by CI/CD pipelines, quality dashboards, and downstream tooling.

5. **Separation of concerns.** Type definitions, specification parsing, validation logic, runtime testing, report generation, and CLI presentation must reside in distinct packages with unidirectional dependencies.

Three of these goals map directly to the SDT axioms: version agnosticism enables Completeness (the same checks apply regardless of specification version), determinism by construction enforces Determinism, and machine-readable output satisfies Observability. The remaining two protect the mapping itself: extensibility lets new principles be added without perturbing the deterministic behaviour of existing checkers, and separation of concerns ensures each axiom's guarantee is enforced—and auditable—in exactly one place. Two classical design patterns [31] realise the first two goals: the *Adapter pattern* for version agnosticism (Section 5.3) and the *Strategy pattern* for extensibility (Section 5.5).

5.1.2 Module and Dependency Structure

DriveBy is implemented as a single Go module (`github.com/meter-peter/driveby/driveby-cli`). The module is organised into eleven internal packages with a strict, unidirectional dependency flow:

```
types <- spec <- loader <- principles <- testing <- engine <- cli
```

No package may import a package to its left. The `types` package has zero internal dependencies, ensuring that data structures can be shared across all layers without introducing cycles. Figure 5.1 visualises this chain for the packages on the critical validation path; Figure 5.2 adds the off-chain `report` package and the external interfaces.

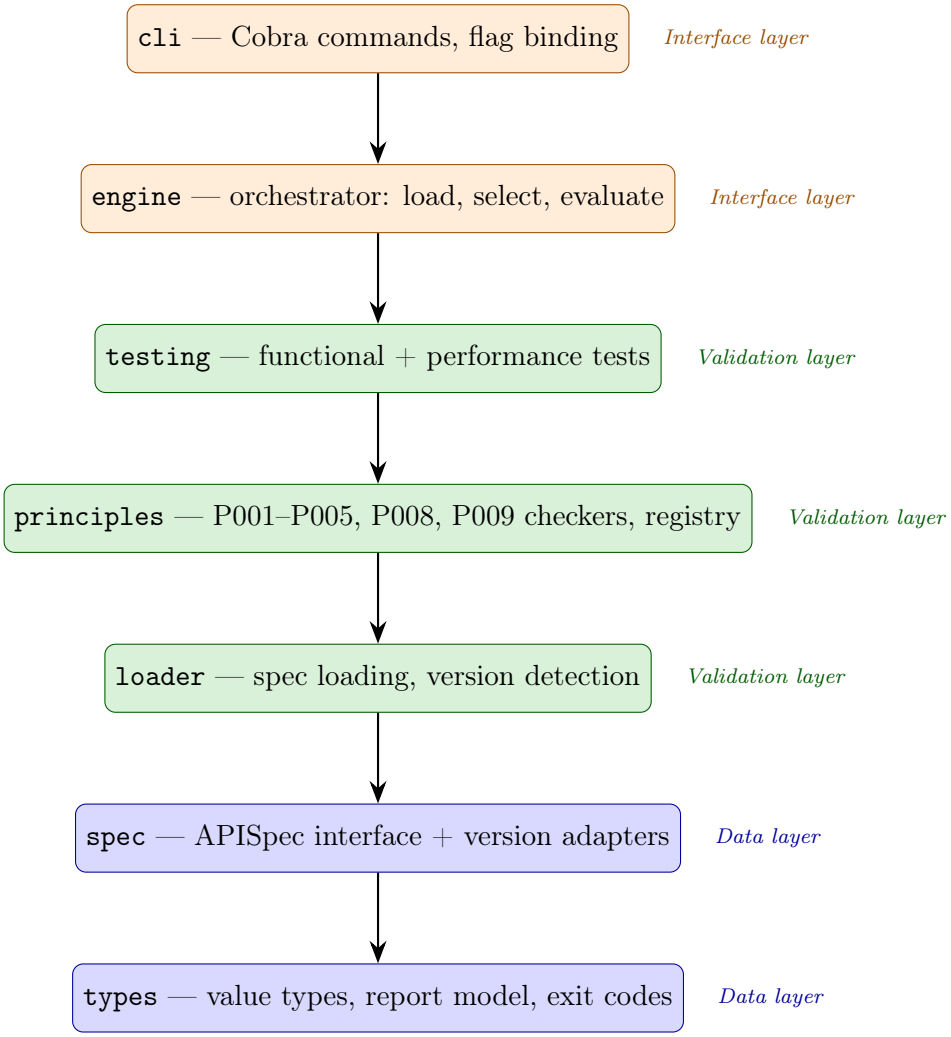

Figure 5.1: DriveBy package dependency graph. Arrows indicate import direction; colour denotes architectural layer.

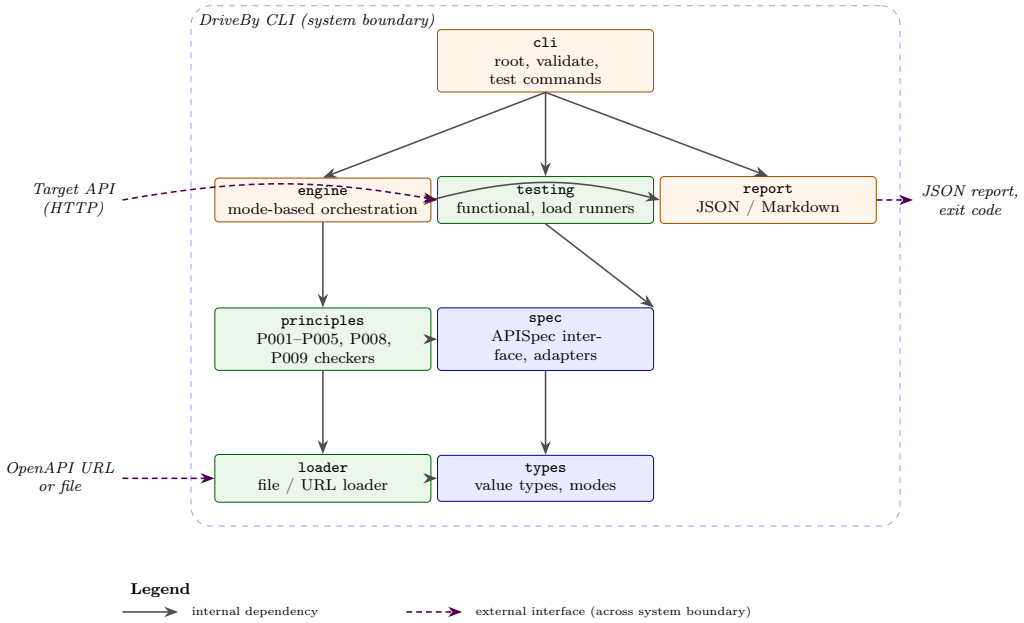

Figure 5.2: Component diagram of the DriveBy CLI. Solid arrows are internal package dependencies; dashed violet arrows cross the system boundary to external interfaces. Colour denotes architectural layer (data, validation, interface), matching Figure 5.1; the off-chain `report` package is assigned to the interface layer. The cross-cutting packages `util`, `github`, and `logger` (Table 5.1) are omitted.

Table 5.1 summarises each package's responsibility and size; the four verbs in the `engine` row name the pipeline steps detailed in Section 5.6.2, and the *registry* in the `principles` row is the checker catalogue of Section 5.5.3.

Table 5.1: Package map of the DriveBy CLI.

Package	Responsibility	LOC
`types`	Value types, report model, exit codes, config	655
`spec`	`APISpec` interface + version-specific adapters	831
`loader`	Specification loading, version detection, preprocessing	322
`principles`	Principle checkers (P001–P005, P008, P009), registry	1,707
`testing`	Runtime functional and performance testing	860
`engine`	Orchestrator: load, select, evaluate, aggregate	117
`report`	JSON and Markdown report generation	670
`cli`	Cobra command tree, flag binding, config assembly	915
`util`	Preprocessing utilities (nullable, exclusive min/max)	81
`github`	GitHub API client: PR comments, commit statuses	917
`logger`	Structured logging initialisation	164

5.2 The Type System

The `types` package defines all value types shared across the framework—the zero-dependency foundation layer.

5.2.1 Value Types

Two core enumerations govern the framework's behaviour: `ValidationMode` controls which principles are evaluated, and the exit code constants control the CLI's return value (Listing 5.1).

The four validation modes correspond directly to the modes defined in Section 3.5 of Chapter 3; the exit code protocol and its operational rationale are detailed in Section 5.9.2.

The `types` package also defines `TestMode` (none, functional, performance, all) and `TestStatus` (passed, failed, warning, skipped, incomplete), which govern the runtime testing subsystem described in Section 5.7.

```go
type ValidationMode string

const (
    ValidationModeStrict   ValidationMode = "strict"
    ValidationModeMinimal  ValidationMode = "minimal"
    ValidationModeTestOnly ValidationMode = "test-only"
    ValidationModeTestReady ValidationMode = "test-ready"
)

const (
    ExitSuccess          = 0  // All checks passed
    ExitValidationFailed = 1  // Validation or tests failed
    ExitExecutionError   = 2  // Runtime error
    ExitInvalidArgs      = 3  // Invalid CLI arguments
)
```

Listing 5.1: ValidationMode and exit code definitions (`types/modes.go`).

5.2.2 Validation Report Model

The `ValidationReport` struct is the central output artefact of every DriveBy run, aggregating per-principle results, summary statistics, and optional test results into one JSON-serialisable structure (Listing 5.2).

The `PrincipleResult` struct captures the pass/fail status, a human-readable message, structured details, a suggested remediation, and the impact on downstream testing. The `TestImpact` field connects static validation to runtime testing, and its gating semantics are deliberate: when a critical principle fails, functional and performance tests do *not* proceed for the affected endpoints, since failures against a contract already known to be unreliable cannot be attributed to the implementation; the assessment tells the pipeline which tests to skip and why.

The `Principle` struct itself carries metadata—ID, name, description, category, severity, tags, and the list of checks—making the report self-documenting.

5.3 The Specification Abstraction Layer

The specification abstraction layer is the architectural centerpiece of DriveBy: it decouples all validation logic from the concrete specification format.

```
type ValidationReport struct {
    Status       string             `json:"status"`
    ExitCode     int                `json:"exit_code"`
    Version      string             `json:"version"`
    Environment  string             `json:"environment"
       `
    Timestamp    time.Time          `json:"timestamp"`
    Principles   []PrincipleResult `json:"principles"`
    TotalChecks  int                `json:"total_checks
       "`
    PassedChecks int                `json:"
       passed_checks"`
    FailedChecks int                `json:"
       failed_checks"`
    Summary      ValidationSummary `json:"summary"`
    TestResults  *TestResults      `json:"test_results
       ,omitempty"`
}

type PrincipleResult struct {
    Principle    Principle
    Passed       bool
    Message      string
    Details      interface{}
    SuggestedFix string
    TestImpact   *TestImpact
}
```

Listing 5.2: ValidationReport and PrincipleResult (`types/report.go`).

5.3.1 The APISpec Interface

The `APISpec` interface defines a version-agnostic view over any API specification. All principle checkers and the engine receive an `APISpec` value; they never interact with the raw types (`*openapi3.T`, `*openapi2.T`) of kin-openapi [32], the third-party parser library underlying the implementation.

```
type APISpec interface {
    Type() SpecType
    RawVersion() string
    Info() *SpecInfo
    Paths() map[string]*PathItem
    Components() *Components
    Security() []SecurityRequirement
    ValidateStructure(ctx context.Context) error
}
```

Listing 5.3: The APISpec interface (`spec/spec.go`).

The interface is intentionally minimal, exposing only what DriveBy's validators need: metadata (`Info`), the operation tree (`Paths`), reusable definitions (`Components`), global security requirements (`Security`), and a structural validation hook (`ValidateStructure`). `Type` returns DriveBy's own `SpecType` enumeration—not a kin-openapi document type, which never crosses the interface boundary—and `RawVersion` returns the verbatim version string; both let callers inspect the flavour when necessary (e.g. P001's null type check), though checkers are strongly discouraged from branching on version.

This design realises the Adapter pattern [31]: supporting a new format (e.g. AsyncAPI) means implementing the interface, with no change to any existing checker.

5.3.2 Normalised Types

The `APISpec` interface returns normalised types that abstract away version-specific representation differences; Table 5.2 lists the key types reached through the methods of Listing 5.3.

The `Schema` type normalises the three composition mechanisms (allOf, oneOf, anyOf) into slice fields, handles nullable semantics across OpenAPI versions, and recursively normalises nested properties and array items, so checkers like P004 can traverse the full schema tree without version-specific logic.

Table 5.2: Normalised specification types in the `spec` package.

Type	Key Fields	Purpose
`Schema`	Type, Properties, Required, AllOf/OneOf/AnyOf, Nullable	Unified schema representation; handles both 3.0.x `nullable` and 3.1.0 `type: ["null", ...]` semantics.
`Operation`	Method, Path, Parameters, RequestBody, Responses, Security	Single HTTP operation with all metadata needed by checkers.
`PathItem`	Parameters, Operations map	Groups operations sharing a path; path-level parameters are propagated.
`Parameter`	Name, In, Required, Schema, Example	Normalised parameter; `In` is a string ("path", "query", "header", "cookie").
`Response`	Description, Content map	Status-code-keyed response with content-type-specific schemas.
`SecurityScheme`	Type, Name, In, Scheme, Description	Auth mechanism metadata for P005 validation.
`Components`	Schemas, Responses, SecuritySchemes	Reusable definitions referenced across operations.

5.3.3 Version-Specific Adapters

Two adapters implement the `APISpec` interface: `openAPI3Spec` for OpenAPI 3.0.x and 3.1.0, and `swagger2Spec` for Swagger 2.0. Both are constructed by the loader (Section 5.4) and wrap the kin-openapi library's [32] native document types.

The OpenAPI 3.x adapter (`openapi3.go`, 324 LOC) performs the bulk of the normalisation work; Listing 5.4 excerpts the function that recursively translates a kin-openapi `*openapi3.Schema` into the normalised `spec.Schema`.

The Swagger 2.0 adapter (`swagger2.go`, 293 LOC) performs analogous normalisation but must account for Swagger-specific conventions: `in: body` parameters become `RequestBody`, security definitions map to `SecurityScheme`, and the single-host model is normalised to the multi-server paradigm. Because kin-openapi provides no `Validate()` for Swagger 2.0, structural validation on this path is a focused set of sanity checks (version discriminator, mandatory `info.title` and `info.version`, at least one path) rather than the full Swagger 2.0 JSON Schema. Load-time validation is therefore knowingly partial: deeper structural defects pass loading but surface through the principle checkers, which operate on the normalised model and apply identically to both formats.

```
func normalizeSchema(s *openapi3.Schema) *Schema {
    if s == nil { return nil }
    out := &Schema{
        Type: s.Type, Format: s.Format,
        MinLength: s.MinLength, MaxLength: s.MaxLength
            ,
        Pattern: s.Pattern, Min: s.Min, Max: s.Max,
        Enum: s.Enum, Required: append([]string{}, s.
            Required...),
        Description: s.Description, Nullable: s.
            Nullable,
    }
    for name, prop := range s.Properties {
        if prop != nil && prop.Value != nil {
            out.Properties[name] = normalizeSchema(
                prop.Value)
        }
    }
    for _, ref := range s.AllOf {
        if ref != nil && ref.Value != nil {
            out.AllOf = append(out.AllOf,
                normalizeSchema(ref.Value))
        }
    }
    return out
}
```

Listing 5.4: Schema normalisation excerpt (`spec/openapi3.go`).

5.3.4 Schema Composition Handling

OpenAPI specifications frequently use `allOf` to compose schemas. The `FlattenAllOf` utility in `spec.go` centralizes the recursive merge: it returns a schema where all properties and required fields from all `allOf` members are combined into a single level. It is used by both the P004 checker and the functional testing subsystem (Section 5.7) when generating example request bodies from composed schemas, avoiding duplicated merge logic across checkers.

5.4 Specification Loading and Preprocessing

The `loader` package is responsible for loading API specifications from files or URLs, detecting the specification format, applying compatibility preprocessing, and constructing the appropriate `APISpec` adapter.

5.4.1 Source Detection and Version Dispatch

The `LoadFromFileOrURL` method detects the source type by path prefix (`http://` or `https://`) and delegates to file reading or HTTP fetching. After obtaining the raw bytes, the loader performs version detection by unmarshalling into a generic `map[string]interface{}` and inspecting two discriminator fields (Listing 5.5):

This approach—unmarshal once into a generic map, inspect a discriminator field, then unmarshal again into the typed struct—avoids speculative parsing with multiple libraries. The preprocessing step is applied only to OpenAPI 3.x specifications and is described in the next section.

5.4.2 Compatibility Preprocessing

The kin-openapi library [32] provides comprehensive support for OpenAPI 3.0.x but has limitations with certain OpenAPI 3.1.0 constructs. DriveBy addresses this through two preprocessing transforms applied to the raw JSON before kin-openapi parses it.

PreprocessNullableAnyOf. OpenAPI 3.1.0 represents nullable types as `anyOf: [{type: X}, {type: null}]`, which kin-openapi rejects. The preprocessor recursively rewrites such two-member `anyOf` arrays to the OpenAPI 3.0.x equivalent: the concrete type with `nullable: true`.

PreprocessExclusiveMinMax. OpenAPI 3.1.0 adopted JSON Schema 2020-12 semantics where `exclusiveMinimum` and `exclusiveMaximum` are numeric values. OpenAPI 3.0.x uses boolean values paired with

```
func (l *Loader) loadFromData(source string, data []
    byte) error {
     var raw map[string]interface{}
     if err := json.Unmarshal(data, &raw); err != nil {
         return fmt.Errorf("failed␣to␣parse␣spec:␣%w",
             err)
     }
     // OpenAPI 3.x: has "openapi" field
     if ver, ok := raw["openapi"].(string); ok && ver
         != "" {
         util.PreprocessNullableAnyOf(raw)
         util.PreprocessExclusiveMinMax(raw)
         processed, _ := json.Marshal(raw)
         doc, err := openapi3.NewLoader().LoadFromData(
             processed)
         if err != nil { return err }
         l.doc = spec.NewOpenAPI3Spec(doc)
         return nil
     }
     // Swagger 2.0 ("swagger" == "2.0"): analogous
         branch
     // constructing spec.NewSwagger2Spec; else error.
     ...
}
```

Listing 5.5: Specification version detection (`loader/loader.go`).

`minimum`/`maximum`. The preprocessor converts numeric exclusive bounds to booleans when paired with their non-exclusive counterparts, and removes them otherwise.

Both transforms operate on the raw `map[string]interface{}` before any typed parsing, preserving the semantic content of the original specification. The approach trades strict 3.1.0 fidelity for broad compatibility—a pragmatic choice given that kin-openapi is the most mature OpenAPI parser available for Go.

5.5 The Principle Checker Pattern

Figure 5.3 summarises the two design abstractions of this chapter—the `APISpec` interface (Section 5.3) and the `PrincipleChecker` interface defined below—together with their concrete realisations and the `Engine` that consumes both.

5.5.1 The PrincipleChecker Interface

Every SDT principle is implemented as a type that satisfies the `PrincipleChecker` interface:

```
type PrincipleChecker interface {
    ID() string
    Check(ctx context.Context, apiSpec spec.APISpec,
          mode types.ValidationMode) types.
             PrincipleResult
}
```

Listing 5.6: The PrincipleChecker interface (`principles/checker.go`).

The interface is deliberately minimal. `ID()` returns the principle identifier (e.g. `"P001"`); `Check()` receives the normalised specification, the validation mode, and a cancellation context, and returns a structured result. The `spec.APISpec` parameter enforces the abstraction boundary: checkers cannot access raw kin-openapi types. This design realises the Strategy pattern [31]: adding a new principle means implementing the interface and registering the checker, with no modification to existing code.

5.5.2 Checker Implementation Pattern

All seven implemented checkers follow a consistent internal pattern. Listing 5.7 shows an abbreviated version of P001 (OpenAPI Specification Compliance) to illustrate the pattern.

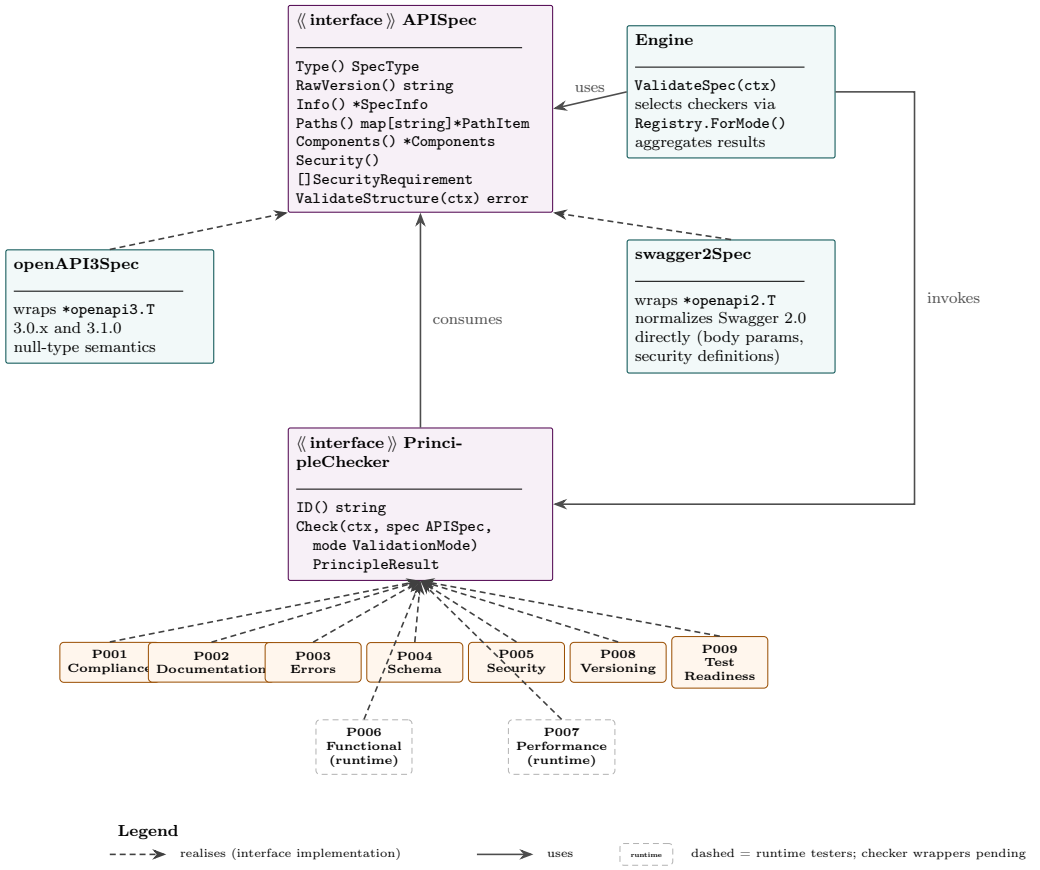

Figure 5.3: Class / interface diagram of DriveBy's two central abstractions. Dashed grey arrows denote interface realisation (concrete class implements interface); solid grey arrows denote use. Dashed nodes (P006, P007) are realised by the runtime testing subsystem (Section 5.7); their `PrincipleChecker` wrappers are pending.

```go
type P001Compliance struct{}

func (p *P001Compliance) ID() string { return "P001" }

func (p *P001Compliance) Check(ctx context.Context,
    doc spec.APISpec, mode types.ValidationMode,
) types.PrincipleResult {
    result := types.PrincipleResult{
        Principle: types.CorePrinciples[0],
        Passed: true, Details: make(map[string]
            interface{}),
    }
    checks := make(map[string]bool)
    // 1. Version check
    if doc.RawVersion() == "" {
        checks["Specification version is present"] =
            false
    }
    // 2. Info fields, 3. Paths, 4. Components ...
    // 5. Structural validation via adapter
    if err := doc.ValidateStructure(ctx); err != nil {
        checks["Specification structure is valid"] =
            false
    }
    // 6. Duplicate operationIds, 7. HTTP methods
    // (null-type semantics are verified within check
        5, structural validation)
    // ... aggregate checks into result ...
    return result
}
```

Listing 5.7: P001 checker pattern (abbreviated, `principles/p001_compliance.go`).

The pattern has four steps: initialise a `PrincipleResult` with metadata from `CorePrinciples`, execute individual checks recording each as a named boolean, aggregate failures into the result, and return it. This consistency simplifies testing (each checker has a corresponding test file in `test/`) and makes the codebase predictable.

Table 5.3 lists all principle checker implementations with their check counts and sizes.

Table 5.3: Principle checker implementations.

File	Principle	Checks	LOC
`p001_compliance.go`	P001: OpenAPI Compliance	7	180
`p002_documentation.go`	P002: Documentation Quality	10	203
`p003_errors.go`	P003: Error Handling	7	245
`p004_schema.go`	P004: Schema Definitions	10	432
`p005_security.go`	P005: Security Standards	7	221
`p008_versioning.go`	P008: Versioning Strategy	7	163
`p009_test_readiness.go`	P009: Test Readiness	4	188

The check counts correspond to those enumerated in Section 3.3 of Chapter 3. P004 is the largest checker because recursive schema traversal must handle nested objects, arrays, allOf compositions, and enums at every level. P009 (Test Readiness) completes the seven-checker set, validating whether the specification provides sufficient examples and typed schemas for meaningful runtime testing.

5.5.3 The Registry and Mode-Based Selection

The `Registry` manages all registered checkers and selects subsets based on validation mode. This centralizes the mode-to-checker mapping defined in Table 3.2 (Chapter 3); Listing 5.8 shows the selection logic.

The `ForMode` method encodes the mode semantics from Section 3.5: minimal mode evaluates only P001; test-ready mode evaluates P001, P002 (test-critical documentation, excluding contact/license), P003 (error schemas for 4xx responses), P004 (type checks only), and P009; strict mode evaluates all six static principles (P001–P005, P008); test-only mode returns no static checkers, deferring entirely to the runtime testing subsystem. When P006 and P007 are implemented as `PrincipleChecker` wrappers, they will be added to the registry and included in the test-only mode selection.

```go
func NewRegistry() *Registry {
    return &Registry{
        checkers: []PrincipleChecker{
            &P001Compliance{}, &P002Documentation{},
            &P003Errors{}, &P004Schema{},
            &P005Security{}, &P008Versioning{},
            &P009TestReadiness{},
        },
    }
}

func (r *Registry) ForMode(mode types.ValidationMode)
    []PrincipleChecker {
    switch mode {
    case types.ValidationModeTestOnly:
        return nil
    case types.ValidationModeMinimal:
        return []PrincipleChecker{r.Get("P001")}
    case types.ValidationModeTestReady:
        return []PrincipleChecker{r.Get("P001"), r.Get
            ("P002"), r.Get("P003"), r.Get("P004"), r.
            Get("P009")}
    case types.ValidationModeStrict:
        return []PrincipleChecker{
            r.Get("P001"), r.Get("P002"), r.Get("P003"
                ),
            r.Get("P004"), r.Get("P005"), r.Get("P008"
                ),
        }
    default:
        return []PrincipleChecker{r.Get("P001")}
    }
}
```

Listing 5.8: Mode-based checker selection (`principles/registry.go`).

5.6 The Validation Engine

5.6.1 Engine Configuration and Initialisation

The `Engine` struct ties together the loader, the registry, and the user's configuration. It is the single entry point for all static, principle-based validation; the runtime testing commands (`function-only`, `load-only`; Section 5.9) bypass it and invoke the `testing` package directly, as Figure 5.2 shows.

```
type Engine struct {
    config   types.ValidatorConfig
    loader   *loader.Loader
    registry *principles.Registry
}
```

Listing 5.9: The Engine struct (`engine/engine.go`).

The `New` constructor validates the configuration (spec path, base URL, positive timeout), defaults the validation mode to `minimal`, and instantiates the loader and registry, so the engine is always in a valid state when `ValidateSpec` is called.

5.6.2 The Validation Pipeline

The core validation pipeline is implemented in a single method, `ValidateSpec`, which executes four steps sequentially (Listing 5.10):

Step 1 delegates to the loader of Section 5.4; step 4 computes the summary statistics—counts of critical issues, warnings, affected categories, and failed tags—that the report generator and CLI use for presentation.

The pipeline is deterministic by construction: given the same specification and mode, the same checkers run in the same order and produce the same results. The only non-deterministic element is the timestamp, which is metadata rather than validation output.

A thin `Validate` wrapper adds logging (structured logs to stderr, JSON report to stdout), preserving backward compatibility with the `APIValidator` component it replaced.

5.7 Runtime Testing Subsystem

While the principle checkers (P001–P005, P008) operate on the specification as a static document, the runtime testing subsystem validates the live API against its specification. The subsystem implements the logic that will

```
func (e *Engine) ValidateSpec(ctx context.Context,
) (*types.ValidationReport, error) {
    // Step 1: Load and normalize the specification
    if err := e.loader.LoadFromFileOrURL(e.config.
       SpecPath); err != nil {
        return nil, fmt.Errorf("failed to load API
           spec: %w", err)
    }
    doc := e.loader.GetDocument()

    // Step 2: Initialize the report
    report := &types.ValidationReport{
        Version: e.config.Version, Environment: e.
           config.Environment,
        Timestamp: time.Now(),
    }

    // Step 3: Select and run principle checkers
    checkers := e.registry.ForMode(e.config.
       ValidationMode)
    for _, checker := range checkers {
        result := checker.Check(ctx, doc, e.config.
           ValidationMode)
        report.Principles = append(report.Principles,
           result)
        if result.Passed { report.PassedChecks++ }
           else {
            report.FailedChecks++ }
    }
    report.TotalChecks = len(checkers)

    // Step 4: Compute summary (critical issues,
       warnings, categories)
    updateSummary(report)
    return report, nil
}
```

Listing 5.10: The validation pipeline (`engine/engine.go`).

underpin P006 (Functional Testing) and P007 (Performance Testing) once their `PrincipleChecker` wrappers are complete.

5.7.1 Functional Testing

The `FunctionalTester` (`testing/functional.go`, 552 LOC) derives test cases from the specification and executes them against a live API. For each non-deprecated operation, it:

1. **Substitutes path parameters** using examples from the specification or type-based defaults (e.g. UUID format generates a valid UUID, integer generates `1`). Example-derived values satisfy the schema's value constraints by construction; generated fallbacks honour type and format but not value-level restrictions (`enum`, `minimum`/`maximum`, `pattern`)—a known limitation of the current generator.

2. **Constructs request bodies** for POST, PUT, and PATCH operations from specification examples or schema-generated values, including recursive `allOf` flattening.

3. **Adds authentication headers** when credentials (Bearer token, API key, or basic authentication) are provided via the root-command flags of Section 5.9.1.

4. **Executes the request** with exponential-backoff retries for transient failures (502, 503, 504).

5. **Validates the response** by checking that the status code is documented in the specification and that the response body conforms to the declared schema (required fields, type checks).

Step 5 is deliberately scoped to *contract conformance*: the status code must be documented and the body must match the declared schema. Whether the returned *values* are semantically correct is outside P006's scope by design—the specification declares structure, not business semantics, so structural conformance is the strongest property checkable without an additional oracle. This is an acknowledged limitation; deriving content assertions from the specification's response examples is noted as future work.

The tester is authentication-aware: if the API returns 401 and no credentials were configured, the endpoint is marked *skipped* rather than failed, preventing false negatives while still reporting the authentication requirement.

All test inputs are derived deterministically from the specification rather than generated randomly, upholding the Axiom of Determinism: the same

specification produces the same test cases and, given a stable API, the same results.

5.7.2 Performance Testing

The `PerformanceTester` in `testing/performance.go` (255 LOC) uses the vegeta library [63] to execute load tests against the API. It constructs targets from non-destructive endpoints—excluding `DELETE` and `PATCH`, retaining primarily safe methods such as `GET`—and runs a configurable load profile: request rate, test duration (`-test-duration`), and concurrency level (`-concurrent-users`). Vegeta calls this an "attack", but only well-formed, specification-derived requests are issued; no penetration testing or security probing is performed. Collected latency percentiles and throughput metrics are compared against user-defined thresholds—maximum p95 latency (`-max-latency-p95`) and minimum success rate (`-min-success-rate`)—producing binary pass/fail outcomes suitable for CI/CD quality gates. Neither profile nor thresholds are embedded in the OpenAPI specification: on the command line they are supplied as the flags above; inside a quality gate they come from the XSDLC `loadTestConfig` block (Chapter 6). The `PrincipleChecker` wrapper for P007 that integrates these results into the unified validation report is pending.

5.8 Report Generation

The `report` package generates validation reports in two formats: JSON for machine consumption and Markdown for human review.

The `Generator` writes each `ValidationReport` as a timestamped JSON file and a timestamped Markdown rendering (per-principle results grouped by category, check lists, suggested fixes), each accompanied by a `validation-report-latest` copy. The timestamped series enables longitudinal comparison across versions or deployments; the "latest" copies provide a stable path for CI/CD scripts. Separate renderers handle functional test reports (per-endpoint results with status codes and response times) and load test reports (latency percentiles, throughput, error rates).

The JSON-first design supports the Axiom of Observability: dashboards, monitoring systems, and downstream pipeline stages consume the structured output without text parsing.

To make this contract machine-checkable, DriveBy publishes a JSON Schema (Draft 07) for the `ValidationReport` format at `driveby-cli/schemas/report.schema.json`. Downstream consumers

can validate report payloads against it, and the schema itself is verified against every sample report under `results/local/` during the release process so that documentation and implementation cannot drift silently.

The JSON-first design also anticipates autonomous engineering agents as consumers: each `PrincipleResult` tells an agent *what* failed (principle ID and checks), *why* (details and message), and *how to fix it* (`SuggestedFix`) [77]. Chapter 9 develops this agent-feedback role of the validation report in full.

5.9 Command-Line Interface

5.9.1 Command Structure

DriveBy's CLI is built with the Cobra framework [30] and organised as a root command with seven subcommands. Table 5.4 summarises the command tree.

Table 5.4: CLI command structure.

Command	Mode	Description
`validate-only`	minimal/strict/test-ready	Run static validation only (P001–P005, P008, P009 depending on mode)
`function-only`	—	Run functional tests only (endpoint contract verification)
`load-only`	—	Run load/performance tests only (vegeta-based)
`test-only`	test-only	Run both functional and performance tests
`github-status`	—	Set GitHub commit status on a specific SHA
`github-comment`	—	Post combined validation report as PR comment
`version`	—	Print version, commit, and build date

Each subcommand follows the same execution pattern: assemble a `ValidatorConfig` from CLI flags (via Viper, which also binds each flag to an environment variable for pipeline configuration), create the appropriate tester or engine, execute, generate reports, write JSON to stdout, and return

an exit code. All logging goes to stderr in JSON format, so stdout contains only the structured report—a critical property for pipeline integration.

The `config.go` file centralizes configuration assembly, preventing flag duplication. Authentication flags (`-auth-token`, `-auth-api-key`, `-auth-username`/`-auth-password`) are defined on the root command and available to all subcommands.

5.9.2 Exit Code Protocol

The CLI uses a four-code protocol that enables CI/CD pipelines to distinguish between different outcomes without parsing the report body. Table 5.5 defines each code.

Table 5.5: CLI exit codes.

Code	Constant	Meaning
0	`ExitSuccess`	All checks passed
1	`ExitValidationFailed`	Validation or tests failed (actionable)
2	`ExitExecutionError`	Runtime error (spec not found, network failure)
3	`ExitInvalidArgs`	Invalid command-line arguments

The distinction between exit 1 and exit 2 is operationally important. Exit 1 indicates that the API or its specification has a quality issue—the pipeline should block promotion but the infrastructure is healthy. Exit 2 indicates an infrastructure problem—the pipeline should retry or alert on-call rather than blocking the developer. This separation reduces alert fatigue by routing different failure modes to different response channels.

Command handlers return errors via Cobra's `RunE` convention rather than calling `os.Exit()` directly. The custom `ExitError` type carries the exit code, and the top-level `Execute` function translates it to the process exit code. This design enables testing of command handlers without process termination.

5.9.3 CI/CD Integration Points

DriveBy is designed to be embedded in CI/CD pipelines as a single binary invocation. Three properties make this practical: (1) *zero configuration defaults*—minimal mode requires only a specification path and a base URL; (2) *stdout/stderr separation*—the JSON report on stdout is consumable without parsing logs; and (3) *GitHub integration*—`github-comment` posts

validation results as PR comments and `github-status` sets a commit status on a specific SHA so branch-protection rules can block merges on failed validation. Chapter 7 describes the pipelines that exercise these integration points.

5.10 Summary

The DriveBy CLI realises the SDT definitions through two central abstractions—the version-agnostic `APISpec` adapter layer and the `PrincipleChecker` strategy pattern—orchestrated by a deterministic four-step engine pipeline, with seven checkers registered (P001–P005, P008, P009) and P006/P007 implemented as dedicated test runners outside the registry. Its zero-dependency types and structured, machine-readable output let the same binary run unmodified in a developer's terminal or a Kubernetes workflow container. Chapter 6 embeds the CLI in declarative quality gates, and Chapter 7 describes the GitOps pipelines that trigger validation and promote environments on its results.

Chapter 6

Kubernetes Architecture

Chapter 5 described the DriveBy CLI as a standalone validation binary. This chapter describes the execution and provisioning layer of the three-layer model (Section 4.2): how that binary is embedded in a Kubernetes-native infrastructure that provisions quality gates declaratively, triggers validation from external events, and orchestrates multi-step validation pipelines as container-native workflows. The result is a system where creating a new API quality gate requires writing a single custom resource—the Kubernetes control plane handles the rest. The architectural evolution that led to this design was traced in Section 4.3.

The architecture draws on four technologies: Crossplane [26] for declarative resource provisioning, Argo Events [10] for event-driven triggering, Argo Workflows [11] for container-native pipeline orchestration, and the GitOps Promoter [34] for automated environment promotion. Each technology addresses a distinct concern in the quality gate lifecycle, and together they realise the ontological state reconciliation pattern introduced in Section 2.4.4 of Chapter 2.

6.1 Architecture Overview

6.1.1 Design Goals

The Kubernetes architecture is shaped by five design goals:

1. **Declarative provisioning.** An operator declares what quality gates should exist; the system provisions all required infrastructure (event sources, sensors, workflows, ingress routes, RBAC) automatically.

2. **Separation of shared and per-API concerns.** Validation logic (the workflow template, RBAC, service accounts) is defined once and shared

across all APIs; event routing (webhooks, sensors, ingress) is defined per API. Changes to the shared validation pipeline propagate to all APIs without per-API updates.

3. **Multi-environment awareness.** The system validates against a source environment (typically development) and gates promotion to a target environment. This inversion—validating the source, not the target—ensures that quality is assessed before deployment, not after.

4. **Progressive configurability.** Sensible defaults enable a minimal-configuration deployment, overridable chart-wide (Helm values) and per-resource (XRD spec fields).

5. **Platform inheritance.** The system assumes shared platform capabilities—ArgoCD, Crossplane, Argo Events, Argo Workflows, Traefik, cert-manager—and references them rather than installing them; they are managed by the platform team and inherited by every `XSDLC` resource.

Figure 6.1 provides a *deployment view* of the cluster the system runs on—it is not a context diagram in the strict SSADM sense (the true context diagram, showing the system's external actors and exchanged artefacts, is Figure 4.1 in Chapter 4). The figure's legend distinguishes components built in this thesis from platform infrastructure that pre-existed on the target cluster (`private.novelcore.org`).

6.2 The Kubernetes Controller Pattern

Every resource described in this chapter—from the `XSDLC` custom resource to individual Sensors and WorkflowTemplates—is managed by a Kubernetes controller that continuously reconciles declared state against observed reality. This section describes the controller pattern, its consistency guarantees, and how Crossplane extends it into a universal control plane.

6.2.1 The Reconciliation Loop

A Kubernetes controller implements a three-phase cycle: *observe*, *diff*, and *act* [44]. It reads the desired state declared in a resource's `spec` and the actual state observed from the cluster (the spec/status pattern of the Kubernetes API conventions [43]), computes their difference, and takes corrective action—creating missing resources, updating drifted configurations, or deleting obsolete objects.

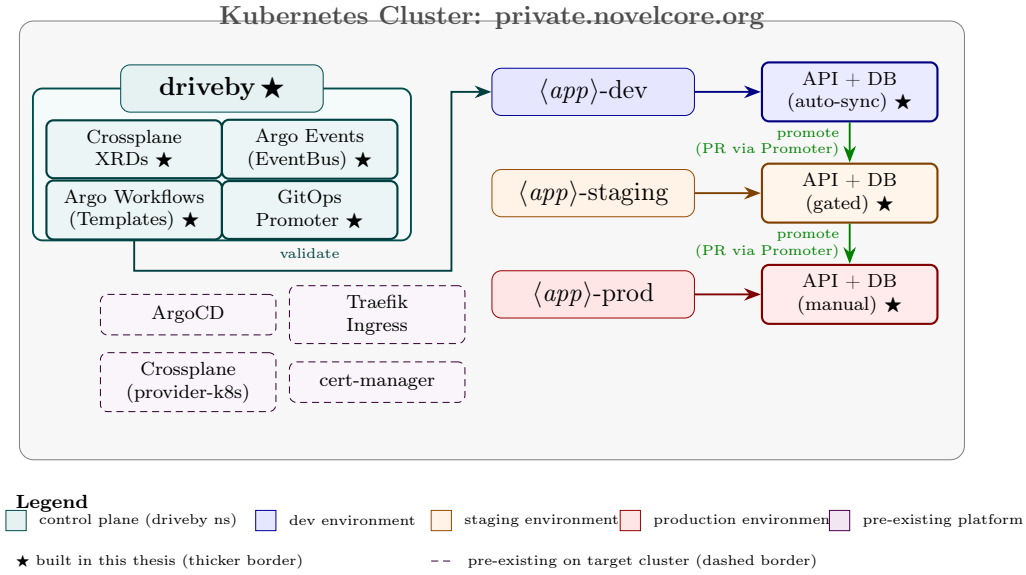

Figure 6.1: Kubernetes deployment view of the target cluster. The `driveby` namespace hosts the quality gate control plane (thesis-built). Per-application namespaces host API deployments across environments (thesis-built, generated per `XSDLC` CR); they are labelled generically as ⟨*app*⟩`-dev`/`-staging`/`-prod`—in the PoC deployment the five application instances used their historical artefact names, e.g. namespaces `perfect-api-dev`, `-staging`, `-prod` for the reference application. Promotion between environments is mediated by the GitOps Promoter (promotion PRs), not by direct environment-to-environment transfer. Shared platform infrastructure—ArgoCD, Crossplane, Traefik, cert-manager—is inherited, not installed. The ★ marker and thicker border indicate components built in this thesis; the dashed border indicates pre-existing platform infrastructure. The strict-sense context diagram of the system as a whole is Figure 4.1 in Chapter 4.

This cycle is *level-triggered*, not *edge-triggered* [16]: the controller reacts to observed state rather than to individual events, so a controller that crashes, restarts, or is scaled horizontally simply re-reads the current state and reconciles from scratch—no missed event can be lost, because the state itself is the complete input. This robustness is fundamental to the self-healing guarantees described in Section 6.2.2.

6.2.2 Eventual Consistency and Self-Healing

Kubernetes does not guarantee immediate consistency. Creating an `XSDLC` resource returns immediately; the composition controller picks it up on its next reconciliation cycle, and the system *converges* toward the declared state over successive cycles, each child resource subject to its own controller's loop. This eventual-consistency model yields self-healing: a deleted child is re-created and a drifted configuration is overwritten on the next reconciliation pass, with no human intervention (concrete scenarios in Section 6.4.3). The model concretely instantiates the ontological state reconciliation pattern of Section 2.4.4 in Chapter 2: the `XSDLC` spec is the ontological artefact, the composition controller the reconciliation agent, and drift correction the correction mechanism.

6.2.3 Crossplane as Universal Control Plane

Crossplane [26] extends the Kubernetes controller pattern by allowing custom resources to declare not just in-cluster objects but any resource reachable through a provider API—cloud databases, DNS records, GitHub branches, and any other resource for which a Crossplane provider exists [25].

The `XSDLC` composition leverages this extension to manage resources across two control planes simultaneously. In-cluster resources (ServiceAccount, EventBus, WorkflowTemplate, ArgoCD Application, secrets) are provisioned through `provider-kubernetes`. GitHub resources (Repository, RepositoryFile, Branch, BranchProtection) are provisioned through `provider-upjet-github`. From the operator's perspective, this distinction is invisible: one `XSDLC` custom resource declares the entire pipeline, and the composition delegates each child resource to the appropriate provider.

Crossplane compositions function as *meta-controllers*: a composition watches a composite resource (the XRD) and generates *desired state for other controllers*—`function-go-templating` renders the desired children, `function-auto-ready` aggregates their readiness, and each child is reconciled by its own provider controller. This two-level reconciliation is what enables a single ∼35-line YAML declaration to produce and maintain a pipeline of

approximately 45–50 managed resources (the exact decomposition is derived in Section 6.4.2).

6.2.4 Provider Architecture

The `XSDLC` composition delegates resource management to two Crossplane providers:

`provider-kubernetes` (v0.14.1) manages all in-cluster resources. It uses `InjectedIdentity` for authentication, inheriting the permissions of the Crossplane pod's service account. Every in-cluster resource generated by the composition (the full inventory is given in Section 6.3.1) is wrapped in a `provider-kubernetes Object` resource, which the provider reconciles by applying the embedded manifest to the cluster.

`provider-upjet-github` (v0.18.0) manages GitHub API resources. It authenticates via a GitHub personal access token stored in a Kubernetes secret. The provider exposes GitHub resources as Kubernetes custom resources: `Repository`, `Branch`, `BranchProtection`, and `RepositoryFile`. The composition generates these resources to auto-create a dedicated GitOps repository, seed it with boilerplate dry manifests, provision environment branches within it, and enforce branch protection rules.

The single-composition, dual-provider model (Figure 6.2) means that the operator interacts with one Kubernetes API (`driveby.io/v1alpha1 XSDLC`) while the system manages resources across two distinct control planes—the Kubernetes cluster and the GitHub API.

6.3 Crossplane XRD Design

The quality gate infrastructure is provisioned through a single Crossplane Composite Resource Definition (XRD): `XSDLC` (API group `driveby.io/v1alpha1`). This single-CR model replaces an earlier two-XRD design, reducing the operator's cognitive load and deployment surface: one custom resource defines the entire software development lifecycle (SDLC) pipeline for an application.

6.3.1 XSDLC: Single-CR Pipeline Definition

The `XSDLC` XRD requires only two fields: `gitopsRepository` (the GitHub owner and gitops repository name) and `environments` (the ordered list of environments in the promotion chain). The *software repository*—the repository holding the application's actual source code—is never referenced; this exclusion is deliberate: XSDLC manages only the gitops repo, leaving build

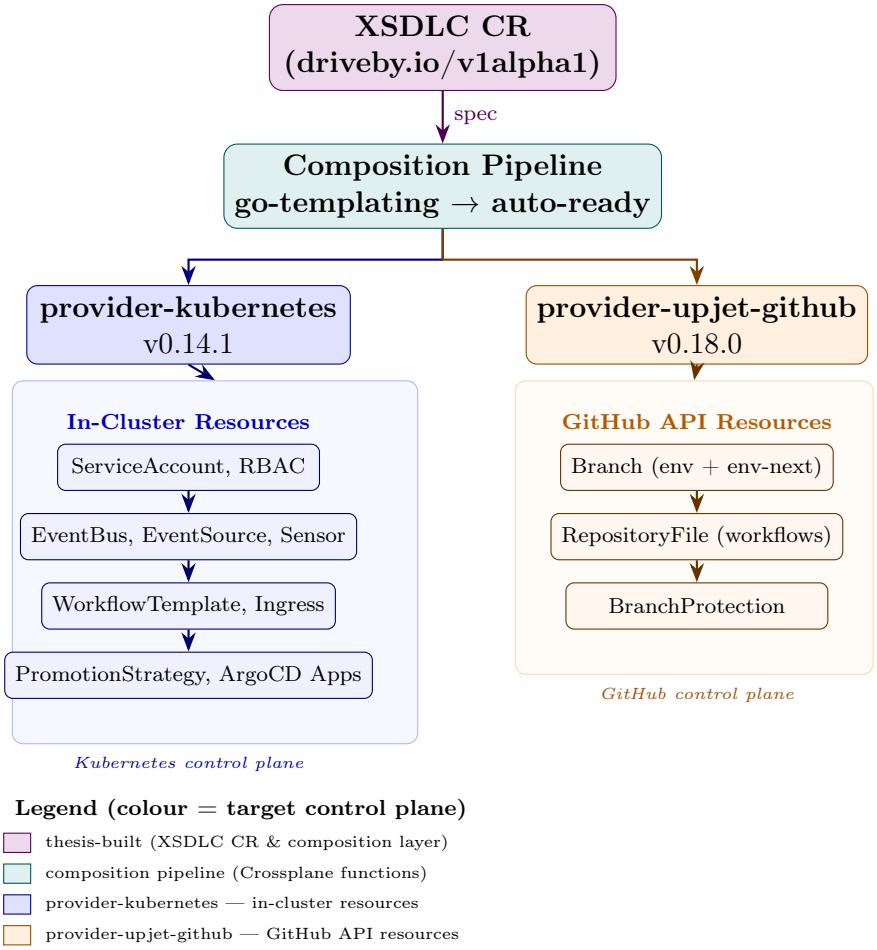

Figure 6.2: Dual-provider architecture. The XSDLC composition delegates in-cluster resources to `provider-kubernetes` (blue) and GitHub resources to `provider-upjet-github` (orange); the composition layer itself (violet) is built in this thesis. Colours encode the target control plane that owns each resource; see the legend at the bottom of the figure.

and CI concerns entirely to the developer's own tooling (the BYOCI boundary, Chapter 7). From these two inputs, the composition generates all resources needed for the full software delivery lifecycle pipeline.

A single `XSDLC` custom resource generates three categories of resources. *App-level shared resources* are created once per CR:

1. A **ServiceAccount** with image pull credentials for accessing the DriveBy container image from the GitHub Container Registry.
2. An **EventBus** backed by NATS JetStream [73] for reliable event delivery between EventSources and Sensors.
3. A **ScmProvider** authenticating to GitHub via a GitHub App.
4. A **GitRepository** pointing to the GitOps repository.
5. A **GitHub Repository** auto-creating a dedicated gitops repository.
6. **RepositoryFiles** seeding the gitops repository with boilerplate dry manifests (base deployment, service, and kustomization).
7. An **ArgoCD push secret** granting the Source Hydrator write access to the gitops repository.[1]
8. A **BranchProtection** rule protecting all `*-next` branches in the gitops repository.

Per-environment resources are created for each environment in the promotion chain:

1. A **PromotionStrategy** defining the environment's auto-merge policy and commit status gates.
2. Two **Branches** in the gitops repository (the environment branch and its `-next` counterpart).
3. A **RepositoryFile** seeding the per-environment Kustomize overlay.
4. A **ghcr-creds** secret copied into the environment namespace for image pull authentication.

[1]The ArgoCD Source Hydrator (Section 6.5.4) renders each environment's Kustomize overlay and *pushes* the hydrated manifests to the corresponding `-next` branch. Unlike ordinary ArgoCD operation, which only reads from Git, this push requires repository-write credentials; the push secret supplies them to the hydrator in the `argocd` namespace.

5. An **ArgoCD Application** using Source Hydrator to render the environment's overlay.

Per-gate resources are created for each environment that defines a `gate`:

1. A **Role** granting permissions to create and manage Argo Workflows, read secrets and configmaps, manage persistent volume claims, and create and update CommitStatus custom resources.
2. A **RoleBinding** associating the service account with the role.
3. A **WorkflowTemplate** containing the validation DAG described in Section 6.8.
4. An **EventSource** configured as a webhook listener on a configurable port (default: 12000).
5. A **Sensor** that filters events (e.g. only `pull_request` events with actions `opened`, `reopened`, or `synchronize`) and submits workflow runs with extracted parameters.
6. A **Service** exposing the EventSource webhook.
7. An **Ingress** with TLS termination (via cert-manager) routing external webhook traffic to the EventSource service.
8. A **BranchProtection** rule requiring the gate's commit status check before merging into the environment branch.

6.3.2 Multi-Check Gate Model

Quality gates in the `XSDLC` model are composed from an ordered `checks[]` array. Each check becomes a sequential step in the Argo Workflows DAG, enabling gates that combine multiple validation concerns in a single promotion decision. Three check types are supported, each mapping to a distinct SDT concern:

- `validate-only` — exploits the DriveBy CLI's static validation facilities, evaluating the specification-level principles, which are applied fully or partially depending on the validation mode: full application in strict mode (P001–P005, P008), partial application in minimal mode (a subset). Accepts an optional `validationConfig.validationMode` override.

- `functional-test` — runs the DriveBy CLI in `function-only` mode, executing P006 functional tests against the live API in the source environment. Validates runtime behavioural conformance: correct response schemas, proper error codes, authenticated endpoint behaviour.

- `load-test` — runs the DriveBy CLI in `load-only` mode, executing vegeta-based load testing (P007) with four configurable parameters and defaults: `concurrentUsers` (10 simulated parallel clients), `testDuration` (`5m`), `maxLatencyP95` (`500ms` 95th-percentile latency ceiling), and `minSuccessRate` (0.99). The defaults are pragmatic baselines: 10 concurrent users over five minutes exercise sustained concurrency without overloading development-grade environments, while the 500 ms P95 and 99% success-rate thresholds reflect common targets for internal HTTP APIs; they were tuned empirically during the PoC deployment and are deliberately conservative, since stricter values can be set per check.

The composition generates DAG steps with deterministic naming (`check-0-validate-only`, `check-1-functional-test`, ...), each depending on the previous, so cheaper checks (static validation) run before expensive checks (load testing), failing fast when possible.

Different environment transitions compose different subsets: the staging gate in Listing 6.1 pairs `validate-only` with `functional-test`, while the production gate pairs `validate-only` with `load-test` (optionally tightened, e.g. `concurrentUsers: 50`, `testDuration: "10m"`, `maxLatencyP95: "200ms"`). This distribution is *indicative*; the actual assignment is chosen by the operator according to the application's requirements.

6.3.3 Minimal Configuration

Listing 6.1 shows a representative `XSDLC` instance. The entire SDLC pipeline for the `perfect-api` application[2]—three environments, a validation gate for staging promotion, and a load test gate for production promotion—is defined in approximately 35 lines of YAML.

```
1 apiVersion: driveby.io/v1alpha1
2 kind: XSDLC
```

[2]Source-code identifiers (namespaces, container images, repository names, the example application) retain the historical name `perfect-api`; the thesis-level rename to `non-critical-api` (see Section 8.3 in Chapter 8 for the rationale) is a naming decision that will be propagated to source in a follow-up release. Wherever this chapter writes `perfect-api`, the same artefact is meant.

```yaml
metadata:
  name: perfect-api
  namespace: driveby
spec:
  gitopsRepository:
    owner: novelcore
    name: perfect-api-gitops
  environments:
    - name: dev
    - name: staging
      gate:
        checks:
          - type: validate-only
          - type: functional-test
    - name: prod
      autoMerge: false
      gate:
        checks:
          - type: validate-only
          - type: load-test
```

Listing 6.1: XSDLC custom resource defining a three-environment pipeline with quality gates.

Note that `validate-only` appears in both gates. Re-running static validation at every promotion gate is intentional and idiomatic: the check is cheap (seconds, no load on the running API), and each gate independently certifies the artefact it promotes—guarding against the specification changing between the staging and production promotions rather than trusting an earlier gate's result.

From this declaration, Crossplane generates all three resource categories of Section 6.3.1; Figure 6.3 visualises the resulting hierarchy. The operator writes approximately 15 lines of meaningful configuration; the composition generates several hundred lines of Kubernetes manifests. All cluster-specific configuration—domain, image references, secret names, ingress class, certificate issuer—is inherited from Helm values, following the two-tier configuration model described in Section 6.6.

Adding a new API to the quality gate system requires creating one `XSDLC` custom resource; no other resources need to be created or modified.

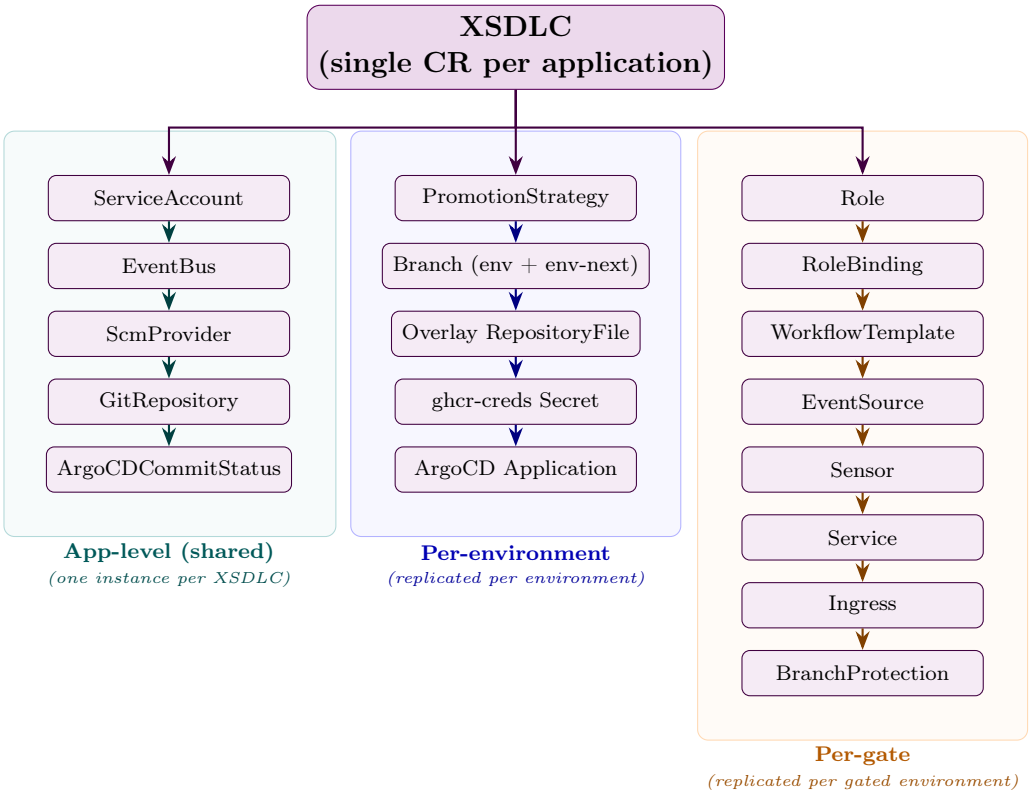

Figure 6.3: Resources generated by the `XSDLC` XRD. App-level shared resources (ServiceAccount, EventBus, ScmProvider, GitRepository, Repository, RepositoryFiles, push secret) are created once per CR. Per-environment resources (PromotionStrategy, Branches, overlay RepositoryFile, ghcr-creds, ArgoCD Application) are created for each environment. Per-gate resources (Role, RoleBinding, WorkflowTemplate, EventSource, Sensor, Service, Ingress, BranchProtection) are created for each environment that defines a quality gate.

6.4 Declarative Lifecycle Management

This section describes how the controller pattern and provider architecture manifest in the operational lifecycle of the quality gate system—from platform installation through pipeline creation, continuous operation, and eventual decommissioning.

6.4.1 Day 0: Platform Installation

DriveBy's own Helm chart (`driveby`, version 3.0.0, published at `oci://ghcr.io/meter-peter/charts/driveby`; Section 6.9) installs the *platform layer*: the XRD definition, the composition template, the ProviderConfig for `provider-kubernetes`, the ProviderConfig for `provider-upjet-github`, and the Crossplane functions (`function-go-templating`, `function-auto-ready`). At this point, no `XSDLC` instances exist. The platform is ready to accept declarations but has not yet provisioned any quality gate infrastructure.

This separation is deliberate: the platform team installs the Helm chart once per cluster, and application teams create `XSDLC` custom resources independently, without platform-team involvement for each new API. The platform provides the *capability*; individual CRs exercise it.

6.4.2 Day 1: Pipeline Creation

When an operator applies an `XSDLC` custom resource, the Crossplane composition controller detects the new resource on its next reconciliation cycle and begins generating child resources. The resource count decomposes along the three categories of Section 6.3.1. *Once per CR*: the GitHub Repository, ServiceAccount, EventBus, ScmProvider, GitRepository, ArgoCD push secret, ArgoCD-CommitStatus, three seed RepositoryFiles, and the `*-next` BranchProtection rule—11 resources. *Per environment*: a PromotionStrategy, two Branches, an overlay RepositoryFile, a ghcr-creds secret, and an ArgoCD Application—6 resources each. *Per gated environment*: Role, RoleBinding, WorkflowTemplate, EventSource, Sensor, Service, Ingress, and BranchProtection—8 resources each. For the reference deployment (`perfect-api` with three environments and two quality gates) this yields $11 + 3 \times 6 + 2 \times 8 = 45$ child resources; across typical configurations (varying environment and gate counts) the total is approximately 45–50, the figure used throughout this thesis. The GitHub-provider resources (Repository, Branches, RepositoryFiles, BranchProtection rules) are generated only when the GitHub provider is enabled; disabling it reduces the count accordingly.

Each child resource is itself subject to its provider's reconciliation loop. The `provider-kubernetes` controller applies each in-cluster manifest; the `provider-upjet-github` controller creates the gitops repository and its branches. The `function-auto-ready` function monitors all children and marks the `XSDLC` composite as `READY=True` only when every child has reached its own ready state.

Figure 6.4 illustrates this reconciliation cycle.

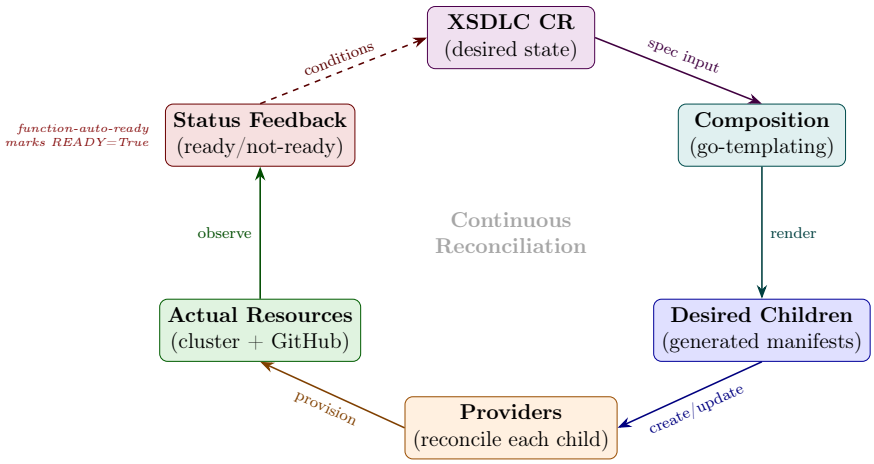

Figure 6.4: Crossplane reconciliation loop. The XSDLC spec feeds the composition, which renders desired children. Providers reconcile each child against its target system. Status feedback flows back through `function-auto-ready` to mark the `XSDLC` composite resource as ready.

6.4.3 Day 2: Continuous Reconciliation

Once the pipeline is created, two scenarios illustrate continuous reconciliation.

Drift detection. If an operator or automated process modifies a Crossplane-managed resource—for example, changing the Sensor's filter configuration—the composition controller detects the discrepancy on its next reconciliation pass and overwrites the change: the `XSDLC` spec is the single source of truth.

Self-healing. If a managed resource is deleted—for example, `kubectl delete sensor perfect-api-staging-gate-sensor`—the composition controller detects the missing child and re-creates it, returning the pipeline to its declared state without human intervention.

The composition uses two deletion policies to distinguish shared and per-pipeline resources. *Shared resources* (ServiceAccount, EventBus) use

`deletionPolicy: Orphan`. If the `XSDLC` CR is deleted, these resources are left in place because other `XSDLC` instances in the same namespace may depend on them. *Per-pipeline resources* (ScmProvider, GitRepository, PromotionStrategy, WorkflowTemplate, EventSource, Sensor, ghcr-creds secrets, ArgoCD push secret, and all per-gate resources) use `deletionPolicy: Delete`. When the `XSDLC` CR is deleted, these resources are cascade-deleted because they are specific to the pipeline being decommissioned. Similarly, GitHub `Branch` and `RepositoryFile` resources use `deletionPolicy: Orphan` to preserve branch history and manifest content even when the pipeline is removed.

6.4.4 Day N: Updates and Deletion

Updates. Modifying an `XSDLC` CR's `spec` field triggers a reconciliation delta: adding an environment generates its ArgoCD Application, branches, and (if gated) per-gate resources; removing one deletes the corresponding children; changing a gate's check types regenerates the WorkflowTemplate. The composition is *idempotent*: the same spec produces no changes, a modified spec exactly the changes needed.

Deletion. Deleting the `XSDLC` CR cascade-deletes all children with `deletionPolicy: Delete`; resources with `deletionPolicy: Orphan` (ServiceAccount, EventBus, Branches, RepositoryFiles) remain, so shared infrastructure survives individual pipeline removal.

6.4.5 Status Reporting

The `XSDLC` XRD defines a structured `status` section that reports the pipeline's operational state. The status fields include:

- `ready` (boolean): overall readiness, set by `function-auto-ready` when all children are healthy.
- `environmentCount` (integer): number of environments in the promotion chain.
- `gateCount` (integer): number of environments with quality gates.
- `gates[]` (array): one entry per gated environment, reporting environment name, gate type, commit status key, webhook URL, and readiness.
- `promotionStrategy` (object): name and readiness of the generated PromotionStrategy.

- `conditions[]` (array): standard Kubernetes conditions (Ready, Synced) with timestamps and messages.

The XRD defines `additionalPrinterColumns` so that `kubectl get xsdlcs` displays a summary view:

```
NAME          APP           ENVIRONMENTS   GATES   READY   AGE
perfect-api   perfect-api   3              2       True    4d
```

This status reporting applies the Observability axiom to the infrastructure itself: just as SDT's validation report makes API quality observable, the `XSDLC` status makes pipeline health observable at a glance, without inspecting individual children.

Figure 6.5 summarises the lifecycle stages.

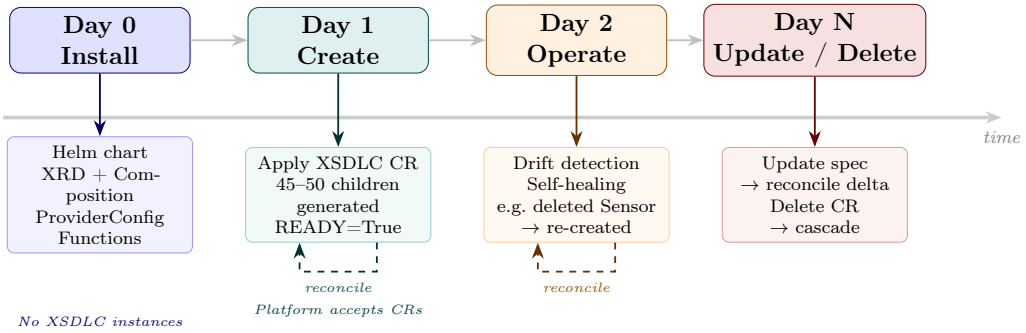

Figure 6.5: Declarative lifecycle management. Day 0 installs the platform; Day 1 creates the pipeline from a single CR; Day 2 and beyond provide continuous reconciliation with drift detection and self-healing; Day N supports updates and clean deletion.

6.5 Composition Pipeline

6.5.1 Pipeline-Mode Compositions

The `XSDLC` XRD uses Crossplane's pipeline-mode composition, which processes the composite resource through a sequence of functions rather than relying on the legacy patch-and-transform approach. The composition pipeline follows a two-step pattern:

1. **function-go-templating**: Renders Kubernetes manifests from Go templates, with access to the XRD spec and Helm-injected defaults.

It performs the bulk of the work: iterating over the environments list and each gate's `checks[]` array, generating shared resources once and per-gate resources (including the step templates `driveby-validate`, `driveby-functional`, `driveby-loadtest`) per gated environment, resolving the two-tier configuration precedence, and handling the template escaping required when generating Argo Workflows templates that themselves contain Go template syntax.

2. **function-auto-ready**: Automatically marks the composite resource as ready when all composed resources reach their ready state.

6.5.2 Template Escaping

A notable engineering challenge arises from the nested templating layers. The Crossplane composition uses Go templates to generate Argo Workflows manifests, which themselves use Go template syntax for parameter substitution (e.g. `{{workflow.parameters.app-name}}`). Without escaping, the Crossplane go-templating function would attempt to evaluate the Argo template expressions during composition rendering, causing failures.

The solution uses a double-escaping convention. Argo template expressions within the composition are written as:

```
{{ "{{" }}workflow.parameters.app-name{{ "}}" }}
```

The outer Go template evaluates `{{"{{" }}` to produce the literal string `{{`, and similarly for the closing braces. The resulting manifest contains the intended Argo expression `{{workflow.parameters.app-name}}` verbatim. This escaping is applied consistently across all workflow step definitions in the template composition.

6.5.3 Generated GitHub Resources

When the GitHub provider is enabled, the `XSDLC` composition generates three categories of GitHub resources, all managed through `provider-upjet-github`.

Branches. For each environment in the promotion chain, the composition generates two branches in the gitops repository, sourced from `main`: `environment/<name>` (the active branch that ArgoCD watches) and `environment/<name>-next` (the proposed branch). Both use `deletionPolicy: Orphan` to preserve Git history when the pipeline is removed. For a three-environment pipeline, this produces six branches.

Repository. The composition auto-creates a dedicated, private GitOps repository under the same GitHub owner (e.g. `novelcore/perfect-api-gitops`), with `deletionPolicy: Orphan`: deleting it would destroy the manifest history and audit trail of environments that may still be running, so decommissioning the pipeline deliberately leaves the Git record intact. The software repository is never modified by XSDLC.

Branch protection. When enabled, the composition generates `BranchProtection` rules for gated environments. Each gated environment's active branch requires the corresponding commit status check to pass before merging. The `-next` branches receive protection to prevent accidental overwrites.

The key insight is that the GitOps repository and its branch topology are *generated infrastructure*, not hand-authored configuration: the composition maintains them in the same reconciliation loop as in-cluster resources, eliminating manual repository setup.

6.5.4 ArgoCD Application Generation

The composition generates an ArgoCD Application for every environment in the promotion chain, not just gated environments. Every Application uses the ArgoCD Source Hydrator [8] rather than a traditional `source` block. Each Application's `sourceHydrator` configuration points to the environment's Kustomize overlay on the `main` branch:

- `drySource`: the gitops repository, path `dry/overlays/<envName>`, revision `main`
- `syncSource`: the environment's active branch (`environment/<name>`), path `manifests`
- `hydrateTo`: the environment's proposed branch (`environment/<name>-next`)

When a developer updates dry manifests on the `main` branch, the hydrator renders the Kustomize overlay for each environment and writes the hydrated output—along with a `hydrator.metadata` file containing the dry commit SHA, satisfying the GitOps Promoter's provenance contract—to the corresponding `-next` branch. The promotion flow that follows (Promoter PR, gate execution, merge, sync from the active branch) is described in Chapter 7.

This per-environment overlay model lets each environment customize its deployment independently (replicas, resource limits, environment variables, image tags), and deploying to a specific environment requires only updating

its overlay on `main`—there is no linear propagation chain through earlier environments.

All generated Applications are configured with `automated.selfHeal: true` and `automated.prune: true`. ArgoCD automatically synchronises changes and self-heals drifted resources regardless of the `autoMerge` setting. The `autoMerge` field exclusively controls the GitOps Promoter's behaviour: when `true` (default), the Promoter automatically merges promotion PRs after gates pass; when `false`, the PR remains open for human review. This decoupling is deliberate—ArgoCD should always reflect the state of the environment branch, while the promotion decision (whether to update that branch) is where human judgment applies.

All generated Applications include `CreateNamespace=true` in their sync options, ensuring that the target namespace is created automatically if it does not exist. Each Application carries labels (`driveby.io/app`, `driveby.io/environment`) that enable the ArgoCDCommitStatus resource to aggregate health signals across all environments.

6.6 Two-Tier Configuration Model

The system implements a two-tier configuration resolution model that balances simplicity with flexibility. Figure 6.6 illustrates the resolution order.

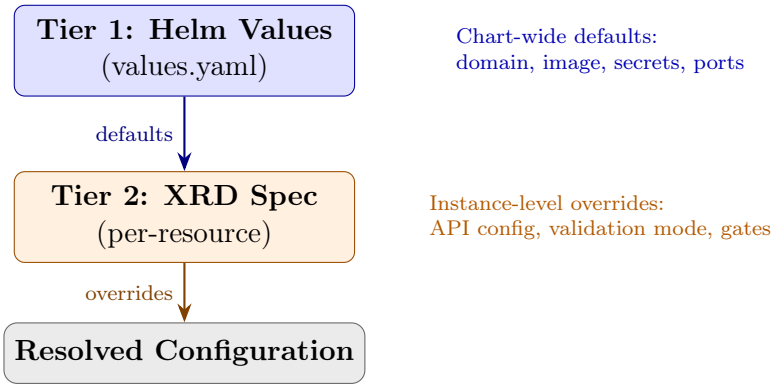

Figure 6.6: Two-tier configuration resolution. Tier 2 (XRD spec) overrides values from Tier 1 (Helm defaults), with the XRD spec having the highest priority.

Tier 1: Helm values (`values.yaml`) provide chart-wide defaults, baked into the composition templates at Helm rendering time. This tier captures all cluster-specific configuration that applies uniformly across all `XSDLC` custom resources.

Tier 2: XRD spec fields provide per-resource overrides with the highest priority, enabling per-application customization (e.g. a different validation mode, a non-standard port, or custom load test parameters).

The resolution is implemented in the go-templating function as a simple precedence check: if the XRD spec provides the value, use it; else use the Helm default. Table 6.1 lists representative parameters and distinguishes the XRD-exposed (per-resource) parameters from the global ones resolved from Helm values only. The elimination of the intermediate EnvironmentConfig tier—present in an earlier three-tier design—simplifies both the mental model and the composition logic without sacrificing meaningful flexibility.

Table 6.1: Representative configuration parameters and their two-tier resolution.

Parameter	**Tier 1 (Helm)**	**Tier 2 (XRD)**
Global (no per-resource override)		
Base domain	`private.novelcore.org`	—
GitHub API URL	`https://api.github.com`	—
Ingress class	`traefik-system`	—
JetStream replicas	1	—
Webhook port	12000	—
Resource-specific (per-resource override)		
API port	8000	per-resource
DriveBy image	`ghcr.io/.../driveby:latest`	per-resource
Load-test thresholds	10 users / `5m` / `500ms` / 0.99	per-check
Validation mode	`strict`	per-resource

6.7 Argo Events Architecture

Argo Events provides the event-driven layer that connects external triggers (GitHub webhooks) to internal actions (workflow submissions). The architecture uses three Argo Events resources, each generated by the `XSDLC` composition for every environment that defines a quality gate.

6.7.1 EventBus

The EventBus is the message transport layer between EventSources and Sensors, backed by NATS JetStream [73] with persistent, at-least-once delivery: events are persisted and replayed if a Sensor is temporarily unavailable, without external message queue infrastructure. The EventBus must be named `default` within each namespace (an Argo Events convention) and runs a single replica in the development deployment; production deployments would scale to three replicas for high availability.

6.7.2 EventSource

The EventSource is the ingress point where external events enter the Kubernetes-native event pipeline: an HTTP server listening for webhook payloads on a configurable port (default: 12000). Webhooks are chosen over polling because quality gate triggers are inherently event-driven, and webhook delivery provides near-real-time notification without polling's latency and API rate-limit pressure.

Each quality gate generates a named webhook endpoint following a deterministic naming convention: `/<appName>-<envName>-gate-pr-trigger` (e.g. `/perfect-api-staging-gate-pr-trigger` for the reference deployment's staging gate). The composition derives the EventSource endpoint, Ingress route, and GitHub webhook URL from the same two identity fields, guaranteeing that they agree without manual coordination.

The EventSource receives the raw webhook payload, validates its structure, and publishes it to the EventBus, decoupling event reception from event processing: filtering and action-triggering are delegated entirely to the Sensor.

6.7.3 Sensor and Parameter Extraction

The Sensor subscribes to events from the EventBus, applies filtering logic, and triggers workflow submissions when filter conditions are met. For pull request validation, the Sensor filters on:

- Event type: `pull_request`
- Action: `opened`, `reopened`, or `synchronize`

This filtering ensures that workflows are triggered only for meaningful PR events—not for label changes, review requests, or other non-code events.

When a matching event arrives, the Sensor extracts parameters from the webhook payload using JSONPath expressions and passes them to the workflow submission; Table 6.2 lists them.

Table 6.2: Parameters extracted from webhook payload by the Sensor.

Parameter	JSONPath Source	Purpose
`app-name`	XRD spec (static)	Identifies the API being validated
`pr-number`	`body.number`	PR comment and status reporting
`head-sha`	`body.pull_request.head.sha`	Commit status target
`github-owner`	`body.repository.owner.login`	Repo owner, addressing commit-status/PR-comment API calls
`github-repo`	`body.repository.name`	Repo name, addressing commit-status/PR-comment API calls
`source-namespace`	XRD spec (static)	Environment to validate against
`target-namespace`	XRD spec (static)	Promotion target
`service-name`	XRD spec (static)	Kubernetes service to validate
`openapi-endpoint`	XRD spec (default: `/openapi.json`)	Spec URL path
`app-port`	XRD spec (default: 8000)	Service port

This hybrid of dynamic webhook values and static XRD-spec values enables the workflow to operate on the specific PR that triggered validation while targeting the correct application environment.

6.8 Argo Workflows Validation DAG

The validation pipeline is implemented as an Argo Workflows WorkflowTemplate containing a directed acyclic graph (DAG), *dynamically generated* by the Crossplane composition from the gate's `checks[]` array (Section 6.3.2); the Sensor submits workflow runs from this template with the extracted parameters.

6.8.1 Pipeline Steps

The DAG follows a fixed preamble, a dynamic checks section, and a fixed postamble:

Preamble (fixed steps):

1. **set-pending-status**: Posts a "pending" commit status to the PR's head SHA via the GitHub API. A parallel step (**update-commitstatus-pending**) creates a CommitStatus custom resource with phase `pending` for the GitOps Promoter.

2. **wait-for-source-ready**: Polls the source environment (e.g. `perfect-api-dev`) with up to 60 `curl` probes of the OpenAPI endpoint at 5-second intervals (5 minutes maximum), ensuring validation does not run against a partially deployed or restarting instance.

Dynamic checks (from `checks[]` array):

For each check in the gate's ordered `checks[]` array, the composition generates a DAG step named `check-<index>-<type>`. Each step depends on the previous step, forming a sequential chain. The step template is selected by check type:

- `validate-only` → `driveby-validate`: Runs the DriveBy container (`ghcr.io/meter-peter/driveby:latest`) with flags derived from the check's `validationConfig`; the validation report is stored on a shared PersistentVolumeClaim (PVC) at `/tmp/reports`.
- `functional-test` → `driveby-functional`: Runs the CLI in `function-only` mode against the live API, with authentication credentials injected from Kubernetes secrets.
- `load-test` → `driveby-loadtest`: Runs the CLI in `load-only` mode, parameterised from the check's `loadTestConfig`; credentials are injected from the same secret, so the generated load exercises authenticated endpoints.

Postamble (fixed steps, parallel after last check):

1. **report-success**: Posts a "success" commit status to the PR's head SHA.
2. **comment-pr**: Posts the validation report as a comment on the pull request, reading saved reports from the shared PVC via DriveBy's `github-comment` command.
3. **update-commitstatus-success**: Updates the CommitStatus custom resource to phase `success`, signaling that the promotion gate has passed.

For example, a staging gate with two checks (`validate-only`, `functional-test`) produces the DAG:

```
set-pending-status → wait-for-source-ready →
  check-0-validate-only → check-1-functional-test →
  report-success / comment-pr / update-commitstatus-success
```

Here → denotes sequential dependency and / separates parallel steps; the `update-commitstatus-pending` step, parallel to `set-pending-status`, is omitted for readability.

6.8.2 Composition Rendering Mechanics

The WorkflowTemplate is the most technically demanding resource in the composition because it operates across *four nested template layers*, each with its own escaping rules and evaluation timing:

1. **Helm layer** (evaluated at `helm template` time). Resolves Helm values: image references, secret names, ingress domains, default ports. Global parameters (e.g. the webhook port, Table 6.1) are baked into the composition at chart installation and cannot change per CR; parameters exposed in the XRD spec merely receive their defaults here and can still be overridden at the next layer.

2. **Crossplane go-template layer** (evaluated at composition reconciliation). Resolves XRD spec fields: application name, environment names, check types, validation modes. The composition uses `function-go-templating` to iterate over the `environments[]` and `checks[]` arrays, generating one WorkflowTemplate per gated environment.

3. **Argo Workflows layer** (evaluated at workflow submission). Resolves workflow parameters: PR number, head SHA, GitHub owner/repo, service name, source namespace. These are injected by the Sensor at runtime from the webhook payload.

4. **Container layer** (evaluated at step execution). Resolves environment variables from Kubernetes secrets: GitHub App credentials, API authentication keys.

Because Argo parameter references share the Go-template double-brace syntax, the composition applies the escaping convention of Section 6.5.2 so that they survive composition rendering verbatim and are evaluated only at workflow submission.

The DAG's dynamic section uses a stateful rendering pattern. The composition maintains a `$state` dictionary with a single key (`prevStep`) that tracks the last generated step name. As the `range` loop iterates over `$checks[]`, each step's `dependencies` field references the value of `prevStep`, and then updates it (Listing 6.2):

```
1 {{- $state := dict "prevStep" "wait-for-source-ready" -}}
2 {{- range $checkIdx, $check := $checks }}
3 {{- $stepName := printf "check-%d-%s" $checkIdx $check.
     type -}}
4 {{- $templateName := "" -}}
5 {{- if eq $check.type "validate-only" }}{{ $templateName =
      "driveby-validate" }}{{ end -}}
```

```
6  {{- if eq $check.type "functional-test" }}{{ $templateName
       = "driveby-functional" }}{{ end -}}
7  {{- if eq $check.type "load-test" }}{{ $templateName = "
      driveby-loadtest" }}{{ end }}
8    - name: {{ $stepName }}
9      template: {{ $templateName }}
10     dependencies: [{{ get $state "prevStep" }}]
11 {{- $_ := set $state "prevStep" $stepName -}}
12 {{- end }}
```

Listing 6.2: Stateful DAG rendering: a sequential dependency chain generated over the configured checks.

This pattern produces a sequential dependency chain of arbitrary length; the postamble steps all depend on the final value of `prevStep`, the last check regardless of how many exist.

Step templates are conditionally included: the composition scans the `checks[]` array (`$needsFunctional`, `$needsLoadtest`) and only emits the step template definitions that are actually referenced, preventing undefined-template errors and keeping the rendered WorkflowTemplate minimal.

Per-check configuration is resolved at DAG generation time: a `validate-only` check passes its `validationConfig.validationMode` (defaulting to the CR's `validationDefaults.validationMode`); a `load-test` check passes `concurrent-users`, `test-duration`, `max-latency-p95`, and `min-success-rate`, each resolved from the check's `loadTestConfig` or, failing that, from the XRD-schema defaults of Section 6.3.2 (10 users, 5 minutes, 500 ms P95, 99% success rate)—defaults, not measurements. Each check in the same gate can thus use different thresholds without duplicating the step template.

A single composition template thus produces WorkflowTemplates of varying structure. Counting every DAG task—the three fixed preamble tasks plus the three fixed postamble tasks—a gate with one `validate-only` check produces a 7-step DAG; a gate with `validate-only` + `functional-test` + `load-test` produces a 9-step DAG with three step template definitions and per-check parameters. The composition handles all variants without conditionals on the DAG structure—the `range` loop and `$state` dictionary suffice.

6.8.3 Exit Handler

The workflow defines an exit handler that triggers on any step failure. It posts a "failure" commit status to the PR's head SHA and, when the GitOps Promoter is enabled, updates the CommitStatus custom resource to phase

`failure`, blocking automatic promotion. The exit handler is critical for the quality gate pattern: without it, a failed workflow would leave the commit status "pending" indefinitely, neither blocking nor approving promotion. It guarantees that every workflow run terminates with an explicit quality signal—success or failure, never ambiguous.

6.9 Helm Chart Packaging

The entire Kubernetes infrastructure is packaged as a Helm chart (`driveby`, version 3.0.0) that installs all Crossplane resources, XRDs, compositions, and provider configuration in a single `helm install` command.

6.9.1 Chart Structure and Install Order

The chart contains Crossplane provider references (`provider-kubernetes` v0.14.1), Crossplane functions (go-templating, auto-ready), the `XSDLC` XRD definition, the `XSDLC` composition, a ProviderConfig for the Kubernetes provider, and optional secret templates.

Installation follows a four-step order designed to handle Crossplane's CRD chicken-and-egg problem: the Crossplane provider must be installed and healthy before ProviderConfig resources that reference it, and the XRD must be installed before the composition that implements it. The recommended sequence is:

1. Pre-install: Crossplane provider and functions (wait for CRDs)
2. Secrets: GitHub PAT, API auth, image pull credentials
3. Helm install: XRD, composition, ProviderConfig
4. Apply `XSDLC` CR: A single custom resource generates all quality gate infrastructure—Roles, RoleBindings, WorkflowTemplates, EventSources, Sensors, Ingresses, ScmProvider, GitRepository, PromotionStrategy, and ArgoCDCommitStatus resources

The ordering of steps 1–3 is not enforced by the chart itself—Helm cannot wait for a provider's CRDs mid-install—so steps 1 and 2 are applied before `helm install`; Listing 6.3 gives the concrete commands.

```
1 # 1. Pre-install: Crossplane provider (repeat for
     functions)
2 kubectl apply -f providers/provider-kubernetes.yaml
3 kubectl wait --for=condition=Healthy \
```

```
    provider/provider-kubernetes --timeout=120s

# 2. Secrets: GitHub App, API auth, image pull credentials
kubectl create secret generic github-app-credentials \
    --namespace driveby \
    --from-literal=githubAppID=<APP_ID> \
    --from-literal=githubInstallationID=<INSTALLATION_ID> \
    --from-file=githubAppPrivateKey=<private-key.pem>
kubectl create secret generic driveby-api-auth \
    --namespace driveby --from-literal=api-key=<API_KEY> \
    --from-literal=api-key-header=X-API-Key
kubectl create secret docker-registry ghcr-creds \
    --namespace driveby --docker-server=ghcr.io \
    --docker-username=<USER> --docker-password=<TOKEN>

# 3. Helm install: XRD, composition, ProviderConfig
helm install driveby \
    oci://ghcr.io/meter-peter/charts/driveby \
    --version 3.0.0 --namespace driveby

# 4. Per application (later, by the application team)
kubectl apply -f xsdlc-<app>.yaml
```

Listing 6.3: Installation commands per phase.

A distinction in scope applies to step 4: steps 1–3 install the *system* once per cluster by the platform team, whereas step 4 is the per-application pipeline declaration (Section 6.4.2), applied later and independently by each application team; Crossplane then provisions all child resources without further operator involvement.

6.9.2 OCI Distribution

The chart is distributed as an OCI (Open Container Initiative) artefact [39], stored in the same container registry that hosts the DriveBy container images. This unified model provides a *single authentication boundary* (one registry credential covers images and charts), *version-aligned storage* (chart and image artefacts sit side by side per release), and *immutability* (OCI tags are treated as immutable, giving tamper resistance and reproducible installations), replacing the traditional separate Helm chart repository. The chart is published to `oci://ghcr.io/meter-peter/charts/driveby`; versioned releases are produced automatically by the release pipeline (Chapter 7) from the same Git tag that versions the container image and CLI binary, keeping all three artefact types version-aligned.

6.10 Summary

This chapter described the Kubernetes infrastructure that transforms DriveBy from a standalone CLI tool into a declarative, event-driven quality gate system, realising the final stage of the architectural evolution traced in Section 4.3. A single `XSDLC` custom resource, reconciled by Crossplane's dual-provider composition under the level-triggered controller pattern, generates and self-heals the full pipeline—GitOps repository, environment branches, ArgoCD Applications, and the Argo Events/Workflows gate machinery with its dynamically rendered validation DAG. Chapter 7 describes how this infrastructure operates within the broader GitOps pipeline: the two-repo architecture, the BYOCI design principle, and the promotion-driven quality gate flow.

Chapter 7

GitOps Pipeline and CI/CD

Chapter 6 described the Kubernetes infrastructure—Crossplane XRDs, Argo Events, Argo Workflows—that implements the quality gate system. This chapter describes how that infrastructure operates within the broader GitOps pipeline: the promotion-driven quality gate flow, the two-repo architecture and its "Bring Your Own CI" (BYOCI) design principle, multi-environment promotion, human-in-the-loop decision patterns, the quality gate as a closed-loop control system, and the automated release of DriveBy itself.

This chapter realises the outermost layer of the three-layer model (Section 4.2)—the point where validation results become operational signals that advance or block software delivery.

The chapter addresses Research Question 3 (RQ3): *How effectively does specification-driven quality assurance integrate into cloud-native GitOps pipelines, and can a single declarative custom resource replace the manual provisioning of quality gate infrastructure?* The answer is demonstrated through a proof-of-concept deployment on the `private.novelcore.org` cluster, where DriveBy validates five evaluation APIs across three environments.

7.1 Promotion-Driven Quality Gate

The quality gate operates on a key insight: *validate the source environment, not the target.* The developer's own CI builds a new container image (the BYOCI boundary described in Section 7.3.2), then commits the updated dry manifests and image tag to the `main` branch of the dedicated gitops repository. The ArgoCD Source Hydrator [8] renders each environment's Kustomize overlay and writes hydrated output to the corresponding `-next` branch. The GitOps Promoter creates a promotion PR from the `-next` branch to the active branch, and ArgoCD [9] synchronises the environment once the PR is

merged.

This promotion PR triggers the quality gate. The GitHub webhook fires, the EventSource receives the event, the Sensor filters for PR actions, and the Workflow executes validation against the *source* (development) environment—the environment where the new code is already running. The validation results determine whether the promotion PR is approved (merged) or blocked (left open with a failure status).

This inversion is deliberate. Validating the source environment ensures that quality is assessed against the actual deployment rather than a hypothetical future state. If the API in development passes strict SDT validation, the same code deployed to staging will behave identically (assuming equivalent infrastructure). The alternative—validating the target environment after promotion—would detect problems only after the potentially breaking change has already been deployed, violating the shift-left principle [68].

Figure 7.1 shows the end-to-end pipeline from deploy trigger to validation feedback.

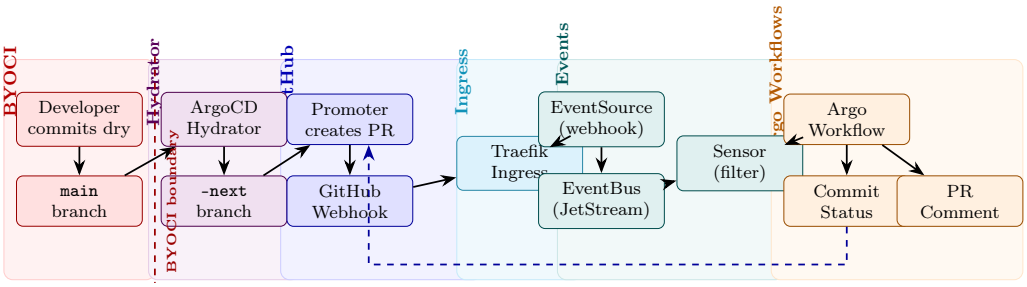

Figure 7.1: End-to-end GitOps validation pipeline. The developer commits dry manifests to the `main` branch of the gitops repository (BYOCI boundary). The ArgoCD Source Hydrator renders per-environment overlays and writes hydrated output to `-next` branches. The Promoter creates a promotion PR; the webhook triggers Argo Events, which submits an Argo Workflow. Validation results are posted back as commit status and PR comment (dashed feedback loop).

7.2 Event-Driven Validation Pipeline

The event-driven pipeline transforms an external webhook into a structured validation workflow. Its components—EventSource, EventBus, Sensor, and the generated Ingress and webhook configuration—are described in Chapter 6; this section covers only their pipeline-level behaviour and latency.

7.2.1 Webhook Delivery and Event Processing

Webhook routing is convention-over-configuration: the composition derives the GitHub webhook URL (`https://{app}-{gate}-webhook.{domain}/{app}-{gate}-pr-trigger`), the Ingress that exposes it, and the EventSource endpoint from the same two identity fields, so they agree without manual coordination (Section 6.1.1, Chapter 6). The Sensor filters for `pull_request` events with action `opened`, `reopened`, or `synchronize`, extracts ten parameters via JSONPath (Table 6.2 of Chapter 6), and submits the generated WorkflowTemplate with those parameters as arguments.

7.2.2 Latency Analysis

The end-to-end latency from webhook delivery to workflow completion depends on three factors: event processing overhead (typically under 2 seconds for EventSource → EventBus → Sensor → workflow submission), source readiness wait (0–120 seconds, depending on whether the deployment is already healthy), and validation execution time (10–60 seconds, depending on specification size, validation mode, and check type—static, functional, or load). In practice, the validation step itself completes in 10–30 seconds: the five-API evaluation (Section 8.4) measured 10–12 seconds for gates with static validation only and 18–28 seconds for gates that include functional or load testing. End-to-end—from PR creation to commit status posted—the proof-of-concept runs show 40–70 seconds as the typical range with a median of 69 seconds (Section 8.2.8), the difference being event-processing overhead and the source-readiness wait; gates whose source environment is unreachable take approximately 5 minutes (the health check retries for 60 attempts at 5-second intervals before timing out).

7.3 Two-Repo Architecture

The quality gate system operates within a *two-repository* model: the software repository is entirely developer-owned and untouched by XSDLC, while a dedicated GitOps repository—auto-created by the XSDLC composition—holds Kubernetes manifests on per-environment branches.

7.3.1 Why a Dedicated GitOps Repository

Traditional GitOps deployments face a tension between two concerns: the developer wants to keep manifests close to the source code for co-evolution, while the platform needs a clean Git-based state store that ArgoCD can watch

without being polluted by source code history, CI artefacts, or unrelated commits. Single-repo models conflate these concerns: every source commit appears in the ArgoCD diff view, and manifest-only environment branches create divergent histories in a repository whose `main` branch carries application source.

The XSDLC v3.0.0 model resolves this tension by separating the two concerns into two repositories. The software repository remains untouched; XSDLC auto-creates a dedicated GitOps repository per application that holds dry Kustomize manifests on the `main` branch and hydrated output on per-environment branches (the layout below uses the reference application's historical source identifier `perfect-api`; see the naming note in Chapter 6):

```
perfect-api/                    # Software repository (untouched)
  src/                          # Application source
  Dockerfile                    # Build specification

perfect-api-gitops/             # GitOps repository (auto-created)
  main branch:
    dry/
      base/                     # Shared manifests
        kustomization.yaml      # refs deployment.yaml + service.yaml
        deployment.yaml
        service.yaml
      overlays/                 # Per-environment customization
        {dev,staging,prod}/kustomization.yaml

  environment/<env>-next     -> # ArgoCD-hydrated (per env)
  environment/<env>          -> # Promoter merges (gate passed)
```

The GitOps repository's `main` branch holds dry Kustomize manifests organised as a base-plus-overlays structure. Each environment has its own overlay that references the shared base and can customize deployment parameters (replicas, resource limits, environment variables, image tags). The ArgoCD Source Hydrator [8] renders each overlay independently and writes the hydrated output—along with a `hydrator.metadata` file containing the dry commit SHA—to the corresponding `environment/<name>-next` branch. The GitOps Promoter detects the new commit and creates a promotion PR from the `-next` branch to the active branch. Quality gates fire on these PRs; ArgoCD syncs from the active branch once the PR is merged.

Per-environment branches and their corresponding `-next` branches are created by the XSDLC composition via `provider-upjet-github`. The software repository is never modified—no workflows generated, no files committed, no branches created—so XSDLC can be adopted without any changes to the developer's existing repository structure, CI configuration, or branching

strategy. Because each environment hydrates independently from `main`, deploying to a specific environment requires only updating its overlay—there is no requirement to propagate changes through earlier environments first.

7.3.2 Bring Your Own CI (BYOCI)

The two-repo model embodies a deliberate design principle: **XSDLC owns delivery, not CI**. The boundary is explicit: *XSDLC's responsibility begins when dry manifests are committed to the `main` branch of the GitOps repository.*

Everything *before* that point is the developer's responsibility: building the application, running unit tests, generating the container image, and updating dry manifests on the `main` branch. These tasks are inherently application-specific—a Python FastAPI service uses different build tools, test frameworks, and container patterns than a Go gRPC service. XSDLC does not generate workflows, provide build templates, or prescribe how dry manifests are authored; the developer can use GitHub Actions, GitLab CI, Jenkins, a shell script, or a manual `git push`.

Everything *after* the commit is XSDLC's responsibility: hydration, promotion PRs, source-environment validation, commit-status-gated merges, and ArgoCD sync. These tasks are platform-generic—the same pattern applies regardless of the application's language, framework, or build tool.

This separation contrasts with opinionated CD platforms (Tekton Pipelines, Jenkins X) that attempt to own the full CI/CD lifecycle. Such platforms trade flexibility for integration: they work well when the application fits their model and poorly when it does not. XSDLC is composable rather than monolithic—it integrates with any CI system that can commit manifests to a Git branch.

7.3.3 Manifest Delivery

The quality gate pipeline is triggered by a single primitive: *a commit to the `main` branch of the GitOps repository that modifies dry manifests.* This is the entire integration contract between the developer's CI and the XSDLC delivery platform. The developer edits the shared base (`dry/base/`) for cross-environment changes or a specific overlay (`dry/overlays/<env>/`) for environment-specific customization. Releasing a new version to staging, for example, is a one-line edit setting `newTag: v1.4.2` in `dry/overlays/staging/kustomization.yaml`; the hydrator re-renders only the staging overlay to `environment/staging-next`, opening a gated staging promotion PR while the other environments are untouched. The developer never creates environment branches, overlay directories,

promotion PRs, ArgoCD Applications, or quality gate infrastructure—the `XSDLC` custom resource provisions all of these.

7.3.4 Branch Topology

Figure 7.2 shows the branch topology in the GitOps repository for a three-environment pipeline.

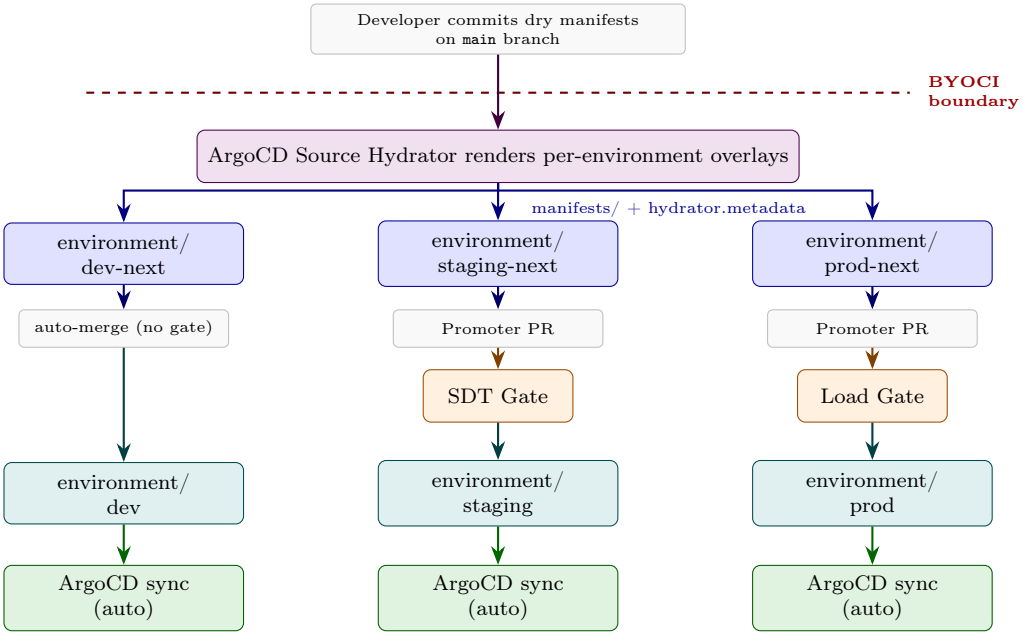

Figure 7.2: Two-repo branch topology. The developer commits dry manifests to `main` in the auto-created GitOps repository. The ArgoCD Source Hydrator renders per-environment overlays and writes hydrated output to `-next` branches. The Promoter creates PRs to active branches; quality gates control promotion (gate names are indicative—each environment configures its own checks); ArgoCD auto-syncs from active branches. The fan-out to all three `-next` branches depicts a shared-base change; a single-overlay commit re-hydrates only its environment. The software repository is untouched—the BYOCI boundary separates developer CI from XSDLC delivery entirely.

ArgoCD watches each active branch and syncs the corresponding namespace from its `manifests/` path.

7.4 Multi-Environment Promotion

The GitOps Promoter [34] automates environment promotion by watching for commit status signals and managing promotion pull requests. Figure 7.3 shows the three-environment model.

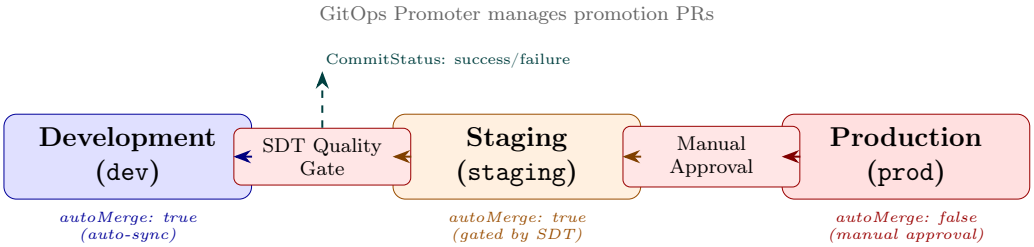

Figure 7.3: Multi-environment promotion model. Development auto-syncs; staging promotion is gated by SDT validation (commit status); production requires manual approval.

The dev → staging → prod chain in Figure 7.3 is an *ordering* over environments, not a propagation path: no hydrated manifest ever moves from the staging branch to the production branch. Each environment hydrates independently from the dry manifests on `main` (Section 7.3.1), and every transition merges that environment's own `-next` branch under that environment's own gate. A change touching only the staging overlay therefore cannot reach production—production's hydrated output is unchanged, so no production promotion PR is created. A change reaches production only by altering production's own hydrated output (via the shared base or the production overlay), and then only through production's gated, manually approved promotion PR.

7.4.1 PromotionStrategy Generation

The PromotionStrategy is not authored manually. The XSDLC composition generates it declaratively from the `environments[]` array in the XRD spec. Each environment entry produces a corresponding element in the PromotionStrategy's environment chain, with three derived fields computed automatically:

- **Branch**: defaults to `environment/<name>` but overridable per environment via `branch`.
- **autoMerge**: defaults to `true`; set to `false` on any environment that requires human approval before promotion.

- **commitStatusKey**: defaults to `<envName>-gate`; overridable via `gate.commitStatusKey`. The key must match between the PromotionStrategy (which watches for it) and the Argo Workflow (which posts it).

The generation model is environment-count agnostic, producing a PromotionStrategy with N environments for any $N \geq 2$ (the composition mechanics are described in Chapter 6).

The key design property is *per-environment composability*. Each environment independently controls three dimensions: *whether* a gate exists (`gate` presence), *what* the gate checks (`gate.checks[]` array), and *how* promotion proceeds after the gate passes (`autoMerge`). A team can run static validation on staging and load testing on production, or functional tests on both, or no gate on any environment—all without modifying the underlying Crossplane composition or Helm chart.

7.4.2 CommitStatus as Quality Signal

The integration between DriveBy and the GitOps Promoter is mediated by two commit status mechanisms:

GitHub commit status. The workflow's set-pending and report-success steps post commit status updates to the PR's head SHA via the GitHub API. The context key (e.g. `staging-gate`) is watched by the PromotionStrategy as the gate signal. This mechanism works for any GitHub-based workflow and does not require the GitOps Promoter.

CommitStatus CRD. When the GitOps Promoter is enabled, the workflow additionally creates a CommitStatus custom resource in the `driveby` namespace. The CommitStatus transitions through three phases: `pending` (set at workflow start), `success` (set after validation passes), or `failure` (set by the exit handler). The Promoter controller watches this resource and uses its phase to determine whether to merge the promotion PR. This CRD-based mechanism provides a Kubernetes-native alternative to the GitHub API and enables the Promoter to function even in environments where direct GitHub API access is restricted.

7.4.3 ChangeTransferPolicy

The ChangeTransferPolicy defines how changes flow between environments at the Git level. For each environment, the policy maps a *proposed* branch (e.g. `staging-next`) to an *active* branch (e.g. `staging`). The Promoter creates pull requests from the proposed branch to the active branch and merges them

when commit status gates pass. This branch-based model integrates naturally with Git-based review workflows: promotion PRs appear in the repository's PR list, carry commit status checks, and leave an auditable merge history.

7.4.4 Promoter Reconciliation Loop

The GitOps Promoter is itself a Kubernetes controller that implements the observe-diff-act pattern (Section 6.2.1) at the Git level rather than the cluster level. It *observes* GitRepository and CommitStatus resources; it *diffs* each proposed branch (`environment/<name>-next`) against its active branch and checks whether all required CommitStatuses report `success`; and it *acts* by creating a promotion PR when the branches differ, merging it when all gates pass and `autoMerge` is `true`, and leaving it open for human review otherwise. Any `failure` or `pending` CommitStatus blocks the promotion.

The ChangeTransferPolicy is itself a second reconciliation layer: it is auto-created by the PromotionStrategy controller when the PromotionStrategy is applied. The PromotionStrategy reconciles the *promotion logic* (which environments exist, what gates are required); the ChangeTransferPolicy reconciles the *branch mechanics* (which branches map to which environments, when to create PRs). This two-layer design separates strategic configuration (the promotion chain) from tactical execution (the branch operations).

7.5 Human-in-the-Loop Patterns

The quality gate system is not fully autonomous. It automates objective assessments—SDT validation, health checks, load testing—while preserving human judgment for subjective assessments. This section maps the automation spectrum and identifies the decision points where human intervention is required or optional.

7.5.1 The Automation Spectrum

The system supports four points on the automation spectrum, controlled by two boolean parameters: `autoMerge` (on the PromotionStrategy, controlling whether the Promoter automatically merges promotion PRs) and `automated.selfHeal` (on the ArgoCD Application, controlling whether ArgoCD automatically syncs and self-heals).

1. **Fully automated** (development): `autoMerge: true`, automated sync. After the developer triggers a deploy, changes flow from the `-next`

branch, are auto-merged by the Promoter, and auto-synced by ArgoCD. No further human intervention after the initial trigger.

2. **Automated with quality gate** (staging): `autoMerge: true`, automated sync, with a commit status gate. The Promoter waits for SDT validation to report success before merging. If validation fails, the PR remains open until the developer fixes the issue and re-pushes. Once the gate passes, promotion and sync are automatic.

3. **Gated with human approval** (production): `autoMerge: false`, auto-sync. The Promoter creates the PR and SDT validates the source environment, but a human must explicitly merge the promotion PR. Once merged, ArgoCD auto-syncs the change. The single human checkpoint is the merge decision—ArgoCD always reflects the branch state.

4. **Fully manual merge**: no quality gate, manual merge, auto-sync. This mode is unused in the reference deployment—it offers no advantage over the gated model—but suits an exploratory or manual-QA environment, where a tester keeps the promotion PR open while exercising the candidate in the source environment and merges when manual testing is complete. ArgoCD still auto-syncs once the branch is updated.

Figure 7.4 illustrates how these two configurable slots—quality gate and merge policy—compose independently per environment; ArgoCD always auto-syncs, so the sync policy is fixed. Adding a quality gate or switching from auto-merge to manual approval are each single-field changes in the XSDLC spec that affect one slot without impacting the other.

7.5.2 Automation with Human Override

The design philosophy is: *automate objective assessments, preserve human judgment for subjective assessments.* SDT validation, health checks, and load testing produce binary signals (pass/fail) based on measurable criteria. These are objective and automatable. Production readiness, business timing, and release coordination involve context that no automated system can capture. These are subjective and require human judgment.

The `autoMerge: false` setting is a deliberate architectural choice, not a limitation. It reflects the principle that the final decision to deploy to production should be an affirmative human act, even when all automated checks have passed. The quality gate provides confidence; the human provides authorization.

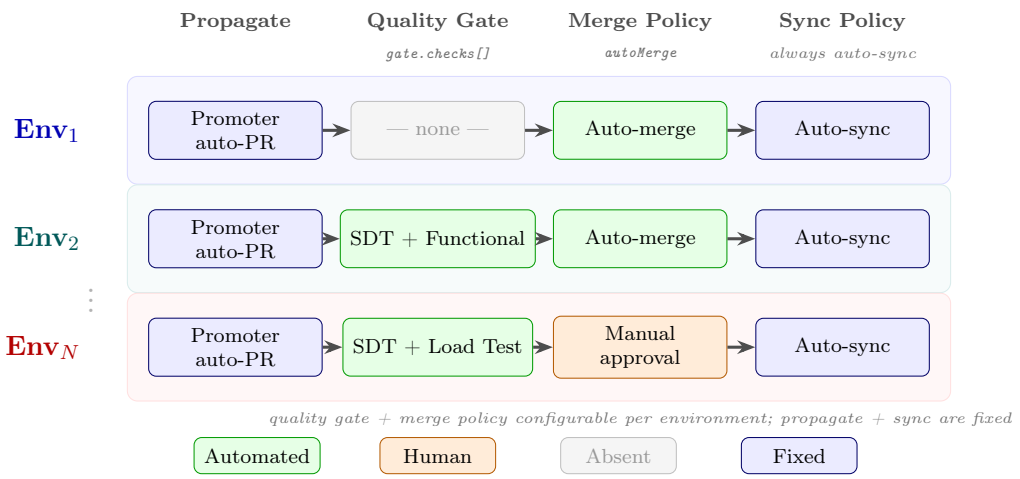

Figure 7.4: Modular automation spectrum. Each environment is a pipeline of four stages with two independently configurable slots: quality gate (`gate.checks[]`) and merge policy (`autoMerge`). ArgoCD always auto-syncs. The composition renders different infrastructure for each combination—from fully automated (Env_1: no gate, auto-merge, auto-sync) through gated automation (Env_2: SDT + functional test, auto-merge) to human-controlled merge (Env_N: SDT + load test, manual approval, auto-sync). The Propagate and Sync stages are fixed; quality gate and merge policy are configurable per environment.

7.5.3 Auditability and Compliance

A consequence of the PR-based promotion model is that every quality gate decision produces a complete, immutable audit trail. Each promotion—whether approved, rejected, or pending—is recorded through multiple complementary artefacts:

1. **Promotion PRs.** Every environment transition is a GitHub pull request (PR). The PR records the proposed change (hydrated manifests on the `-next` branch), the target branch, the timestamp, and the actor who created it (the Promoter). For `autoMerge: false` environments, the PR also records who approved and merged it.

2. **Commit status checks.** Each gate writes a GitHub commit status (`success` or `failure`) on the PR's head SHA. Branch protection rules require these checks to pass before merge is permitted. The commit status persists as a permanent record of the gate verdict.

3. **PR comments.** The exit handler posts a structured DriveBy report as a PR comment: per-principle pass/fail, severity breakdown, remediation guidance ("How to Pass This Gate"), and a link to the Argo Workflow run. Failed gates explain which principles blocked promotion and why.

4. **CommitStatus CRDs.** The Kubernetes-native `CommitStatus` custom resource (managed by the GitOps Promoter) records the gate phase (`pending`, `success`, `failure`) for each SHA. These CRDs persist in the cluster and can be queried for historical gate decisions.

5. **Workflow logs.** Argo Workflow archives contain the full execution trace: health check attempts, DriveBy validation output (JSON), functional test endpoint results, and load test metrics. These logs are accessible via the Argo Workflows UI and persist according to the cluster's retention policy.

This multi-layer audit trail is an intrinsic byproduct of the GitOps promotion model, not a separate observability system: for any deployment in any environment, an auditor can trace back to the exact PR, the exact validation results, and the exact human (or automated) decision that approved the promotion. The `autoMerge: false` setting on production environments adds an explicit human authorization step to this trail—the PR comment provides the evidence ("6/6 principles passed, load test P95: 4.9 ms"), the human makes the merge decision—satisfying the separation-of-duties requirement common in regulated environments.

7.6 Quality Gate as Control Loop

The quality gate system forms a *closed-loop negative feedback* controller [12]. The plant is the *delivery system itself*—the promotion pipeline, whose state comprises promotion PRs, environment branches, and per-environment sync state. The setpoint is a *promotion-policy predicate* over artefacts and delivery state, defined by two declarative artefacts: the OpenAPI specification declares the contract a deployed artefact must satisfy before it may transition, and the XSDLC manifest declares which transitions are gated, by what checks, and under what merge policy. DriveBy measures the deployed API and emits a *validation verdict*; the actuator (CommitStatus and Promoter) merges or blocks the promotion PR, so that only policy-conforming transitions occur. This section first presents this block-diagram mapping, then decomposes its abstract "Controller" into the two-tier stack the implementation instantiates—a verdict tier emitting the measurement and a platform tier performing continuous closed-loop repair of the pipeline machinery.

7.6.1 Control Theory Mapping

Table 7.1 maps each control-theoretic role to its concrete implementation in the DriveBy architecture.

Table 7.1: Control theory mapping to DriveBy quality gate components.

Control Role	Implementation	Function
Setpoint	Promotion policy (OpenAPI spec + XSDLC manifest)	Allowed-transitions predicate over artefacts and delivery state
Plant	Delivery system (pipeline + promotion workflow)	Promotion PRs, environment branches, sync state
Sensor	DriveBy CLI	Measures the deployed API (source env)
Controller	Argo Workflow DAG	Evaluates measurement against policy, emits pass/fail
Actuator	CommitStatus + Promoter	Merges or blocks promotion PR—an action on the delivery system
Measurement	Validation verdict	Structured report + commit status, fed back to the controller

The key property is that the feedback is *negative*: validation failure blocks promotion rather than advancing it. Without this, a failing API would propagate unchecked from development to staging to production—the shift-right antipattern that SDT explicitly prevents. Under this mapping the loop is genuinely closed on its plant: the merge/block decision is a corrective action *on the delivery system*, admitting only artefacts that satisfy the policy predicate. The loop deliberately does not repair the API or its specification; Section 7.6.2 develops that boundary and decomposes the abstract "Controller" into the stack of reconcilers that realises it on the cluster.

Figure 7.5 illustrates the closed loop.

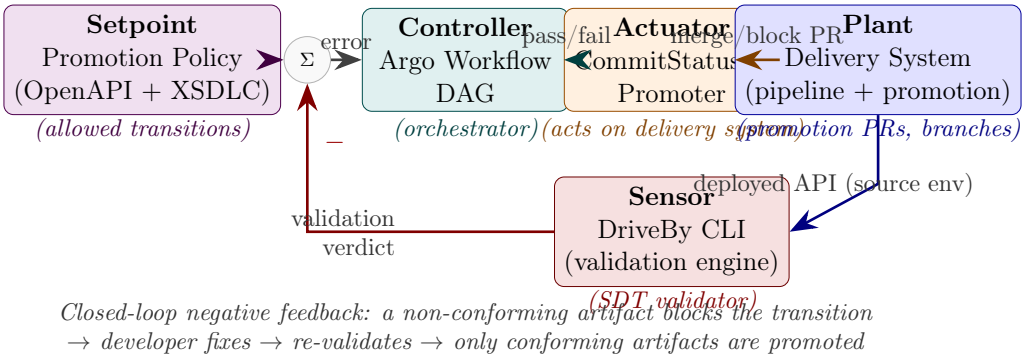

Figure 7.5: Quality gate as a classical control loop. The promotion-policy predicate (OpenAPI spec + XSDLC manifest) is the setpoint; the delivery system is the plant; the DriveBy CLI is the sensor, its validation verdict the measurement; the Argo Workflow is the controller; CommitStatus and Promoter form the actuator that merges or blocks the promotion PR. Negative feedback blocks non-conforming transitions. The platform tier is collapsed into the single "Controller" block; Section 7.6.2 and Figure 7.6 decompose it into the stack of CRDs and reconcilers that realise it.

7.6.2 The Controller is a Stack: Verdict Tier and Platform Tier

The classical mapping in Table 7.1 and Figure 7.5 is pedagogically useful but architecturally lossy: it collapses what is in fact a *stack of controllers* into a single "Controller" block. The system separates cleanly into two tiers with different control-theoretic character.

Verdict tier (open-loop). The verdict tier contains the DriveBy CLI as sensor and the Argo Workflow as the controller that emits a pass/fail signal materialised as a `CommitStatus`. The controller observes the error $e(t) = r_{\text{OpenAPI}}(t) - H(y(t))$, where H is the DriveBy validator and y is the deployed API. It decides *promote* or *block*. This tier is honest about its limitation: *it produces no actuation signal u that reduces $e(t)$ by acting on the API or its specification*—its only actuation is on the delivery system (the plant of Table 7.1), via the merge/block decision. There is no spec re-write, no implementation patch, no automated remediation. The error signal is exported as a structured report; the corrective action is performed externally—by a developer who reads the report, or, in the experiment of Section 8.6, by an AI coding agent that consumes the same report as machine-actionable input. The verdict tier is the deliberate, scoped open-loop component of an otherwise self-reconciling system; closing it with autonomous remediation is

recorded as future work (Chapter 11, Section 11.5).

Platform tier (closed-loop, autonomous). Beneath the verdict tier sits the second reference signal: r_{XSDLC}, the XSDLC manifest, which declares the entire delivery pipeline—gate workflows, event sources, sensors, branch protection, the GitOps repository, the ArgoCD Application, the PromotionStrategy, ingress routes, RBAC. This reference is reconciled by a *stack* of Kubernetes controllers, each with its own observe-diff-act loop in the sense of Section 6.2.1. Three of them perform continuous state reconciliation in the strict sense: Crossplane composes the ∼45–50 managed objects from the manifest and re-creates any that are deleted; ArgoCD reconciles cluster state against the GitOps repository (with `selfHeal` enabled); the GitOps Promoter reconciles promotion branches and reopens PRs that go missing; `provider-upjet-github` reconciles GitHub-side state, restoring branch protection rules and push secrets if they are removed. Two further controllers manage event-time machinery rather than persistent state: Argo Workflows executes the gate as a DAG and retries failed steps under its own retry policy, and Argo Events reconciles the EventSource and Sensor CRDs that keep the webhook plumbing armed. Crossplane plays the role of meta-orchestrator over the stack: the XSDLC composition is what creates and parameterises every controller-managed object in the first place. Figure 7.6 draws the decomposition.

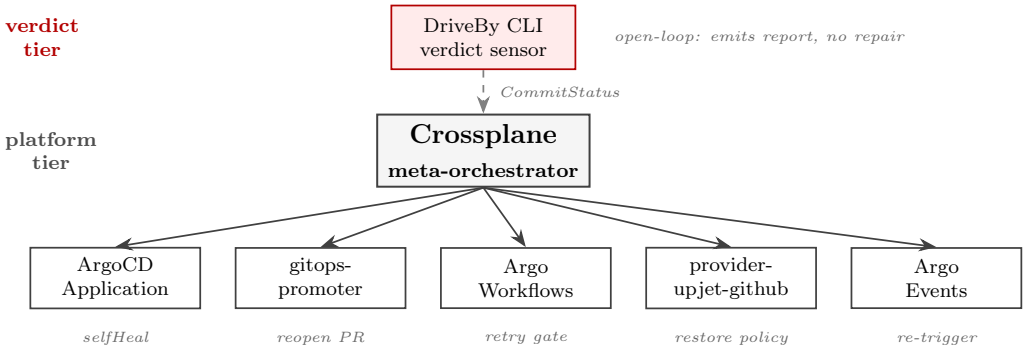

Figure 7.6: The controller stack. The verdict tier (DriveBy) is open-loop: it emits a structured report but performs no repair on the API or the spec. The platform tier is closed-loop and autonomous: Crossplane composes the managed objects from the XSDLC manifest and orchestrates a stack of controllers, each performing continuous reconciliation within its CRD domain. The captions under each platform-tier box name the dominant repair action each controller performs when its declared state diverges from observed state.

This two-tier reading sharpens what "self-reconciling" actually means in

this thesis. The platform tier exhibits the classical closed-loop properties of Kubernetes controllers [44]: drift between r_{XSDLC} and observed cluster state is detected and corrected without human intervention; transient failures of individual controllers do not produce permanent state divergence because the reconciliation is level-triggered. The verdict tier deliberately does not have these properties on the API contract—automatic spec or implementation repair is a research direction, not a present capability—but it inherits the platform tier's robustness for everything *around* the verdict: missing gate workflows are recreated, deleted commit statuses are re-issued, drifted RBAC is restored.

The architectural value of the two-tier separation is precisely that it forces the open-loop component to be small, scoped, and visible. The system is honest about where it intervenes autonomously (platform configuration) and where it relies on a human or agent loop closure (specification and implementation quality).

Drift between specification and implementation deserves explicit treatment in this context, because it is sometimes incorrectly held to be a pre-condition that SDT must externally assume, and a continuous drift-watching loop is sometimes proposed as a missing capability. Both framings understate the architecture. P006 (Functional Testing) detects spec–implementation divergence *at the moment the gate fires*—it probes the live API and compares observed behaviour against the declared contract—making drift detection a first-class capability of the verdict tier rather than an assumption.

The deeper point is that under the XSDLC promotion chain, *the pipeline actively prevents drift from reaching production*—it does not merely detect it after the fact. ArgoCD synchronises a target environment exclusively from its `environment/<env>` branch; that branch is updated exclusively by Promoter-mediated merges from `<env>-next`; the Promoter blocks the merge as long as any required CommitStatus is not in the success state; CommitStatuses are written exclusively by gate workflows that run the principle checkers, including P006. GitHub-side branch protection, reconciled by `provider-upjet-github`, blocks direct pushes to the `environment/*` branches, preventing the chain from being bypassed at the source-control layer. There is therefore no path by which an implementation change can reach production without first passing a gate that has executed P006 against it—and blocked promotion if divergence was detected. The pipeline does not lack continuous drift detection; it makes continuous drift detection unnecessary by enforcing, at every transition, the very property that continuous detection would later have to recover. The only path by which undetected drift could appear in a production deployment is an out-of-band intervention that bypasses the pipeline altogether—a violation of the GitOps discipline that the architecture is designed to enforce, and a

problem that precedes drift detection rather than being solved by it.

7.6.3 Convergence and Robustness

Convergence has a distinct character in each tier. In the *verdict tier*, each quality gate evaluation is a discrete sample: pass or fail. When validation fails, the developer (or agent) fixes the issue and re-triggers the deploy workflow. The new manifests land on the `-next` branch, the Promoter updates the PR, and the gate re-evaluates. The loop iterates until the API conforms to its specification—convergence is driven by external corrective action, gated by DriveBy's objective assessment. The verdict tier converges over PRs, not over time within a PR.

In the *platform tier*, convergence is autonomous and continuous. The Kubernetes controller model (Section 6.2.1) makes the platform tier robust to transient failures by construction. Two failure classes are distinguished. *Transient infrastructure failures*—pod eviction, network timeout—are absorbed within the workflow run: Argo Workflows retries the failed step, so validation completes without developer involvement. If a workflow fails terminally, the exit handler posts a `failure` status and promotion remains blocked (fail-closed); the same change can be re-evaluated without a new commit—the Sensor's `reopened` filter re-fires the gate against the unchanged `-next` branch when the promotion PR is closed and reopened (or the workflow is resubmitted from the Argo UI). *Genuine validation failures* require a corrective commit by design: the re-push is the corrective action itself and re-triggers evaluation through the normal hydration path. If a generated child resource is deleted, the Crossplane composition controller re-creates it on the next reconciliation pass. If a branch protection rule is removed manually, `provider-upjet-github` restores it. No manual re-triggering is needed because the system is level-triggered: it reacts to current state, not to events it might have missed.

The two convergence modes compose. The platform tier guarantees that the gate machinery itself remains in its declared configuration even under operator error or transient infrastructure failure; the verdict tier then operates on top of that guarantee, with developer-driven (or agent-driven) iteration the only loop that requires external action.

7.6.4 Connecting Axioms to Control Requirements

This control-theoretic framing connects to the ontological state reconciliation pattern introduced in Section 2.4.4: each tier has its own ontological artefact and its own reconciliation agent. The OpenAPI specification is the ontological

artefact for the verdict tier, with DriveBy as reconciliation sensor and the developer (or agent) as the external corrective actor; the XSDLC manifest is the ontological artefact for the platform tier, with Crossplane and the downstream controllers as the autonomous reconciliation stack.

The three SDT axioms map to necessary control-loop properties. *Completeness* requires that the setpoint fully characterises the desired behaviour—an incomplete specification produces an under-determined loop that cannot distinguish conforming from non-conforming APIs. *Determinism* requires consistent sensor output—a non-deterministic validator would oscillate between pass and fail for identical states. *Observability* requires structured, actionable output—without it, feedback degenerates into a binary gate with no diagnostic value, and developers cannot identify what to fix.[1]

This is not a metaphorical mapping: each component—setpoint, sensor, controller, actuator, feedback path—has a concrete implementation, and the two-tier decomposition is faithful to the architecture. The correspondence explains why the platform tier converges autonomously, why the verdict tier requires external loop closure today, and why the axiomatic foundation is necessary rather than merely desirable—giving future work on autonomous remediation a precise object of study.

7.7 CI/CD of DriveBy Itself

DriveBy's own development pipeline serves as a reflexive demonstration of the SDT framework: the tool that validates API quality is itself subject to automated quality gates at every stage of delivery. The pipeline design follows three principles: (1) a *test gate* that blocks all artefact production—no binary, image, or chart is published unless tests pass; (2) *parallel artefact generation*—independent outputs (binaries, container images, Helm charts) are produced concurrently to minimise release latency; and (3) *multi-format*

[1]This control-theoretic framing is illustrative rather than formally rigorous. The system does not satisfy the classical control-theory definitions of *stability* (convergence of the error signal under a defined metric), *controllability* (state reachability), or *observability* (state reconstructibility from outputs); the use of the latter term in the SDT axiom set is deliberately coincidental in language but distinct in content—SDT's Observability axiom refers to structured, machine-readable output, not state-reconstructibility. What the framing does capture is the structural *pattern* of closed-loop feedback control: a setpoint (the specification), a sensor (the validator), a controller (the gate workflow), and an actuator (the promoter) that together drive the system toward the declared state and self-correct deviations. A fully formal control-theoretic treatment would require additional assumptions about API behaviour and validator determinism that exceed the scope of this thesis; it is recorded as a research direction in Chapter 11, Section 11.5.

distribution—a single Git tag produces version-aligned artefacts across three distribution channels, ensuring that consumers always reference a consistent release. Three GitHub Actions [33] workflows implement these principles.

7.7.1 Continuous Integration

The `ci.yml` workflow implements a layered validation strategy: compilation verifies build integrity, static analysis detects code-level defects, testing validates behavioural correctness, and a documentation check—an SDT-adjacent meta-validation—fails the build when CLAUDE.md files and documentation entries diverge from the source tree. Although the cheapest layer, the documentation check deliberately runs last: a documentation-divergence report against broken code would be noise—the code, not the documentation, must change first.

The workflow uses a two-tier triggering strategy: pull requests run all four validation layers with coverage artefacts for fast feedback, while pushes to `main` additionally build the container image (without pushing), maintaining the stricter invariant that every commit on `main` is both tested and container-buildable.

7.7.2 Docker Image Publishing

The `docker-publish.yml` workflow uses path-scoped triggering: it fires only when files under `driveby-cli/` change on pushes to `main`, so the container image is rebuilt exactly when the validation logic changes and not otherwise.

The workflow produces images with a dual-tagging strategy: the mutable `latest` tag serves development clusters and Argo Workflows templates that should always run the newest version, while the short commit SHA tag provides an immutable reference for production traceability. Docker Buildx [27] handles multi-platform builds with GitHub Actions layer caching. The resulting image at `ghcr.io/meter-peter/driveby` is the same image used by the Argo Workflows validation pipeline.

7.7.3 Release Pipeline

The `release.yml` workflow implements a fan-out pattern: a single semantic version tag push (`v*`) triggers a test gate, and three independent artefact producers run in parallel after the gate passes. Figure 7.7 shows the pipeline structure.

The test gate re-runs static analysis and the test suite; all subsequent jobs depend on it. This is the release-level analog of the quality gate pattern: just

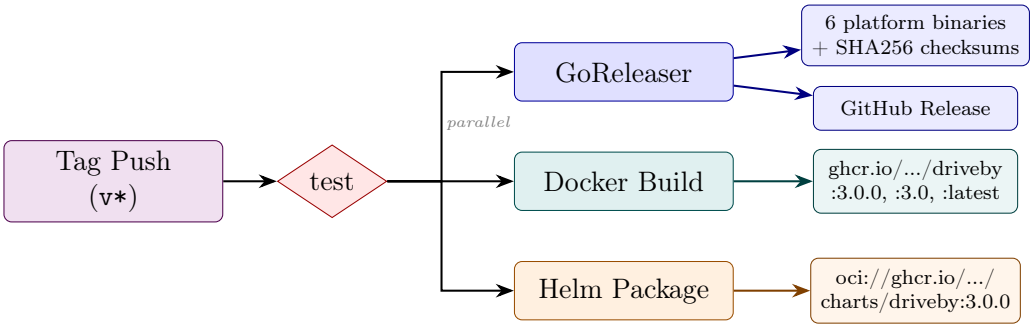

Figure 7.7: Release pipeline triggered by a version tag push. After the test gate passes, three parallel jobs produce platform binaries, Docker images, and a Helm chart.

as SDT blocks promotion on validation failure, the test gate blocks artefact production on test failure.

The three parallel producers each address a distinct distribution channel: GoReleaser [15] produces cross-compiled platform binaries with SHA256 checksums and an auto-generated changelog; the Docker job produces container images with a semver-derived tagging strategy (full version, major.minor, `latest`) that lets consumers choose their stability-versus-freshness trade-off; and the Helm job packages the chart and pushes it to the OCI registry (Section 6.9.2).

The key design property is *version alignment*: the chart version, container image tags, and CLI binary version are all derived from the same Git tag, so consumers can verify artefact provenance without cross-referencing build logs. Parallel execution minimises release latency while the shared test gate ensures atomicity: either all three artefacts are produced from tested code, or none are.

7.8 Operational Deployment

The quality gate system follows a *platform-inherited* deployment model: it does not install its own infrastructure but instead declares references to shared platform capabilities that are assumed to be present. This is a deliberate architectural choice that aligns with the separation between Day 0 (platform installation) and Day 1 (pipeline creation) described in Section 6.4.1. The proof-of-concept deployment runs on `private.novelcore.org`, a Kubernetes cluster managed by KubeCore.

7.8.1 Prerequisites

The deployment depends on six cluster components, each addressing a distinct concern in the quality gate lifecycle. Crossplane [26] provides the declarative provisioning layer that translates `XSDLC` custom resources into managed child resources. ArgoCD [9] provides the GitOps synchronisation layer that deploys applications from environment branches. Argo Workflows [11] and Argo Events [10] provide the event-driven execution layer that triggers and orchestrates validation pipelines. Traefik and cert-manager provide the network ingress layer that routes external webhooks and terminates TLS.

The DriveBy Helm chart creates resources that *reference* components at each layer (Ingress resources reference the Traefik ingress class, certificate annotations reference the cert-manager issuer, workflow submissions reference the Argo Workflows controller) without installing them, so DriveBy can be added to any cluster that already runs these components without conflicting with existing installations.

The proof-of-concept deployment on `private.novelcore.org` validates this surgical coexistence model. The cluster hosts Novelcore's Internal Developer Platform (IDP), with ArgoCD, Argo Workflows, Argo Events, and Traefik already running in production for other workloads; GitHub is the SCM provider because Novelcore already operates on GitHub. DriveBy was installed into this *existing, active cluster* without modifying any pre-existing installation. The same property holds in reverse: deleting an `XSDLC` CR removes all DriveBy-managed child resources (WorkflowTemplates, Sensors, EventSources, Ingresses, BranchProtection rules) while leaving the platform components untouched, and `helm uninstall` removes the XRD and composition definitions themselves. Because DriveBy never owns the infrastructure it depends on, installation and removal require no restart, reconfiguration, or redeployment of any platform component.

7.8.2 Verification

Post-deployment verification is the *platform operator's* responsibility and proceeds layer by layer, validating each abstraction level independently to isolate failures. Steps 1–3 run once after applying an `XSDLC` CR (Day 1), with step 3 also exercising the ingress and certificate components installed at platform installation (Day 0); step 4 exercises the steady-state path (Day 2) that ordinary PR commits subsequently trigger:

1. **Declarative provisioning**: All `XSDLC` composites report `READY=True`, verifying that Crossplane generated and reconciled all child resources; a

non-ready composite indicates a composition rendering error or provider authentication failure.

2. **Event infrastructure**: The EventBus, EventSource, and Sensor resources are healthy; a failing EventBus typically indicates a NATS JetStream configuration issue, a failing Sensor a filter or template reference error.

3. **Network path**: A test webhook payload to the Ingress endpoint returns 200, confirming the path from the external domain to the EventSource; a failure isolates the problem to DNS, TLS provisioning, or ingress routing.

4. **End-to-end behaviour**: A test PR triggers a complete validation cycle—workflow completion, commit status on the PR, and a posted validation report confirm that all layers function together.

7.9 Summary

This chapter described the operational GitOps pipeline that connects DriveBy's validation engine to a fully declarative delivery lifecycle: the BYOCI boundary (Section 7.3) leaves CI to the developer, while a single `XSDLC` custom resource (∼35 lines of YAML) provisions hydration, promotion PRs, per-environment quality gates, and a tunable human-in-the-loop merge policy (Section 7.5). The control-theoretic framing (Section 7.6) explains why the design converges: negative feedback blocks non-conforming promotions, and the three SDT axioms—Completeness, Determinism, Observability—are necessary conditions for the loop to function. Chapter 8 evaluates this system empirically, measuring SDT's effectiveness across controlled and large-scale settings.

Chapter 8

Evaluation

This chapter evaluates SDT and the DriveBy framework across four arms, the first three ordered by increasing breadth: the operational proof-of-concept (the system actually running and gating promotions on Novelcore's internal platform, concluding with the CI/CD integration metrics that quantify its footprint), the controlled experiments against a reference API suite designed to exercise all nine principles, and the large-scale evaluation against real-world public APIs from the APIs.guru registry [6]. The fourth arm—the agent-feedback experiment (Section 8.6)—closes the loop by testing whether the framework's structured validation output is, by itself, sufficient feedback for an autonomous agent to improve a defective specification, demonstrating that the signals the system produces are actionable.

8.1 Evaluation Framework

The evaluation addresses the four research questions stated in full in Chapter 1:

RQ3 (GitOps integration; declarative gate provisioning) Evaluated via the proof-of-concept deployment on Novelcore's cluster (Section 8.2), capturing exact resource counts, gate execution volume, edge-case behaviour, and the CI/CD integration metrics (Section 8.2.8). RQ3 is addressed first: the PoC establishes that the system runs end-to-end before the other arms characterise its detection behaviour.

RQ1 (feasibility of automated, tester-free promotion-gate quality assurance) Evaluated via the single-API controlled experiment (Section 8.3, including the per-check and combination extensions of Section 8.3.4), the multi-API controlled evaluation (Section 8.4), the large-scale evaluation

(Section 8.5), which generalises the detection results to a population of public APIs, and the agent-feedback experiment (Section 8.6).

RQ2 (automatic derivation of validation rules; actionable machine-readable signals) The derivation half is evidenced by the operational deployment (Section 8.2) and the large-scale experiment on 50 public APIs (Section 8.5), which derive all validation rules from the specification with zero hand-authored test cases. The actionability half is evidenced primarily by the agent-feedback experiment (Section 8.6), in which the validation report alone drives autonomous specification correction, and secondarily by the agent-driven analysis workflow of Section 8.5.1.

RQ4 (pipelines as Kubernetes-native ontological specifications) Addressed through architectural analysis in Chapter 6 and operationally validated by the multi-application coexistence observed in Section 8.2.

Figure 8.1 illustrates the four-arm evaluation design in the order the chapter develops it.

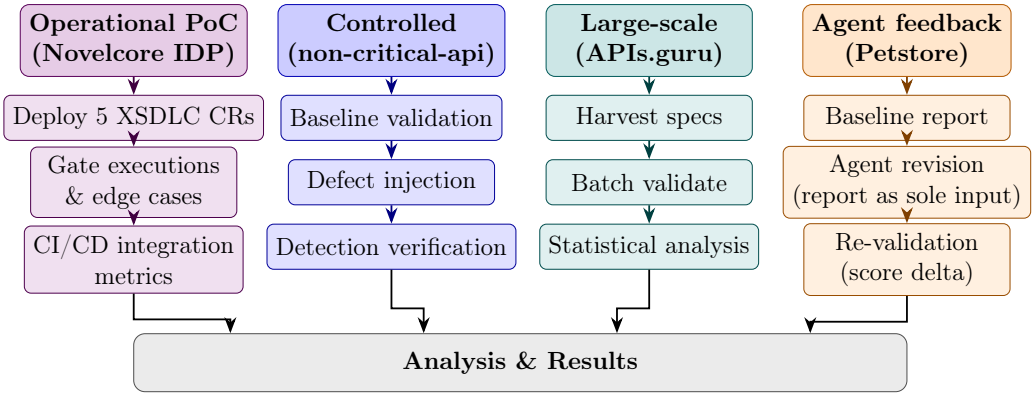

Figure 8.1: Four-arm evaluation design, ordered as presented in this chapter: (1) proof-of-concept operational deployment on an internal developer platform, (2) controlled validation against a reference API suite, (3) large-scale validation against public APIs from APIs.guru, and (4) the agent-feedback experiment in which the validation report is the sole input to autonomous specification improvement.

8.2 Operational Evaluation: Proof-of-Concept Deployment

The operational evaluation assesses SDT not as an isolated tool but as a proof-of-concept (PoC) feature within an Internal Developer Platform (IDP), proceeding from the pre-existing platform state (Section 8.2.1) through the PoC configuration, observed deployment metrics, and operational behaviour to the CI/CD integration metrics (Section 8.2.8).

8.2.1 Initial Situation

Before any PoC artefact was deployed, the target cluster (`private.novelcore.org`) already hosted Novelcore's KubeCore-branded IDP, a Kubernetes platform layer that pre-existed this thesis. The relevant pre-existing components were: ArgoCD (cluster-wide GitOps controller), Crossplane v1.16 with provider-kubernetes, Argo Events with a NATS-based EventBus, Argo Workflows, Traefik as the ingress controller, and cert-manager wired to a public-CA `ClusterIssuer`.

Outside the cluster, five GitHub repositories under the `novelcore` organisation existed for the five evaluation applications (`perfect-api`, `bad-docs-api`, `no-auth-api`, `slow-api`, `broken-api`); in line with the two-repository BYOCI model (Chapter 7), each application repository contained only application source code and the BYOCI workflow that builds container images.

The PoC therefore began against a stable IDP baseline. The thesis-built additions are scoped to the `driveby` namespace (Crossplane XRD, Composition, RBAC, EventBus, WorkflowTemplates, GitOps Promoter installation, and the `XSDLC` resources themselves) and to the per-application namespaces that each `XSDLC` CR generates. No modification to the pre-existing platform components was required; this is the principal architectural property the operational evaluation is designed to demonstrate.

8.2.2 Proof-of-Concept Configuration

DriveBy installation. The DriveBy Helm chart was installed into the dedicated `driveby` namespace (its contents are catalogued in Chapter 6); separately, the GitOps Promoter controller (a gate-architecture dependency, not part of the chart), the GitHub provider credentials, and the ArgoCD push secrets for the auto-created GitOps repositories were installed. Everything else in the `driveby` and per-application namespaces is not installed manually but generated by the Composition when an `XSDLC` resource is applied.

Per-application resources. With DriveBy installed, five `XSDLC` custom resources were applied to the cluster, one per evaluation application. Each CR is 33–39 lines of YAML and shares a common environment chain—`dev` → `staging` → `prod`—differing only in the *gate configuration* on the staging and production transitions. Table 8.1 captures the experiment's gate configuration.

Table 8.1: Proof-of-concept configuration: gate checks per application per environment, reproduced from the `XSDLC` manifests in `kubernetes/examples/`. All five applications share the dev → staging → prod chain and carry `autoMerge:false` on the production transition; differences are confined to the gate checks.

Application	**Staging gate**	**Production gate**
`perfect-api`	validate-only (test-ready) + functional-test	load-test (10 users, 15 s, P95 ≤ 2 s, success ≥ 50%)
`bad-docs-api`	validate-only (strict)	validate-only (strict) + load-test (default thresholds)
`no-auth-api`	validate-only (strict)	validate-only (strict) + functional-test
`slow-api`	validate-only (strict) + functional-test	validate-only (strict) + load-test (10 users, 15 s, P95 ≤ 200 ms, success ≥ 99%)
`broken-api`	validate-only (strict) + functional-test	validate-only (strict) + load-test (default thresholds)

Two design choices are worth flagging. First, the staging gate for `perfect-api` (now `non-critical-api`—see Section 8.3) uses `test-ready` mode rather than the `strict` default for its validate-only check. `test-ready` runs P001–P004 and P009 (excluding P005 and P008 from the strict-mode set); `non-critical-api` clears both modes, so the choice has no operational effect but exercises the mode-selection mechanism end-to-end. Its production gate carries only the load-test check: static validation has already passed at staging against a byte-identical specification, so the production gate adds the one check staging did not run. Second, all five applications carry a production gate with `autoMerge:false`, so no promotion to production can complete without an explicit human merge; the production gates of `bad-docs-api`, `no-auth-api`, and `broken-api` were never exercised because all three are blocked at the staging gate by design (Section 8.4.4).

8.2.3 Activity Volume Captured During the PoC

Cluster-wide and per-application snapshots were captured at the end of the PoC window using `kubectl get` and persisted in `results/cluster/` and `results/edge-cases/`; Table 8.2 reports the activity volume they record as exact counts, not approximations.

Table 8.2: Activity volume captured at PoC end across all five applications. Sources: `results/cluster/all-*.json`, `results/cluster/<app>/*`.

Resource	Source snapshot	Count	Scope
ArgoCD Applications	`all-argocd-apps.json`	15	3 envs × 5 apps
Argo Workflow runs	`all-workflows.json`	44	gate executions across the PoC window
GitOps Promoter CommitStatus CRDs	`all-commitstatuses.json`	32	per-commit per-environment
Approximate Git commits per gitops repo	inspection	6–12	promotion PRs + hydrator commits

The 32 CommitStatus CRDs decompose by phase as 18 `failure`, 13 `success`, 1 `pending`—consistent with the four defective APIs being intentionally blocked at the staging gate while `perfect-api` and `slow-api` progressed further. The counts differ because every gate execution—including the deliberate retries of the workflow re-trigger edge case (Section 8.2.7)—creates a *new* Workflow object, whereas each commit SHA per gated environment receives exactly one CommitStatus CRD updated in place; the surplus of 12 Workflow runs over the 32 commit/environment pairs corresponds to those retries.

8.2.4 XSDLC as IDP Feature

The DriveBy quality gate system is deployed on `private.novelcore.org` as one composable building block of Novelcore's Crossplane-based IDP, following the same declarative pattern as the platform's other capabilities (Chapter 6). A single `XSDLC` custom resource (∼35 lines of YAML) provisions **approximately 44–47 Kubernetes objects** (depending on counting convention;

the compositional derivation in Chapter 6 yields 45 for this reference configuration and approximately 45–50 across typical configurations) in under 15 seconds for a 3-environment pipeline. For the `perfect-api` CR observed on the cluster, the count decomposes as: 3 ArgoCD Applications, 2 WorkflowTemplates, 4 Argo Events resources, 2 Ingresses, 5 GitOps Promoter resources, per-namespace ServiceAccount/RBAC and secrets, and—when the GitHub provider is enabled—branches, BranchProtection rules, and boilerplate manifests in the auto-created GitOps repository (the full generated-resource catalogue is given in Chapter 6). The 44–47 spread reflects whether the per-environment branch and protection objects are counted once or per environment.

Without XSDLC, a platform operator would create each of these resources manually through ClickOps across the ArgoCD UI, GitHub settings, and `kubectl`—a process taking hours, error-prone, and impossible to version-control. XSDLC reduces it to a single declarative resource reviewed in a pull request and reconciled automatically.

8.2.5 Quality Assurance Without Dedicated Testers

The most consequential operational finding is that SDT demonstrates the feasibility of operating without dedicated API testers in the promotion pipeline. Because validation is embedded in the promotion flow—every promotion PR is automatically validated against the specification and, where configured, load-tested—the gate runs without human initiation, test authoring, or result interpretation. One qualification applies: where `autoMerge:false` is configured (as on all five production gates in this PoC), a human must still merge the promotion PR—the validation *verdict* is produced without human involvement, but the final promotion *decision* remains a deliberate human act. This proof-of-concept deployment validates the architectural viability of zero-tester promotion gates, though production adoption would require further hardening and organisational validation.

In the proof-of-concept deployment, API owners can verify that their deployments have been validated against two contracts:

1. **The specification contract** (P001–P005, P008, P009): The API's documentation, error handling, schema definitions, and security configuration match the declared specification. Any *structural* drift is detected before promotion; *semantic* drift—structurally valid but substantively wrong response content—lies beyond the static contract and is caught only to the extent that functional testing (P006) exercises the affected paths.

2. **The performance contract** (`loadTestConfig`): the API meets the configured concurrency, P95 latency, and success-rate thresholds; any regression below them blocks promotion.

This is not a claim that SDT replaces all testing. Unit tests, integration tests, and exploratory testing remain the developer's responsibility (the BYOCI boundary). SDT replaces the *promotion-gate testing* that would otherwise require a dedicated QA engineer before each environment transition.

8.2.6 Load Testing as Business Insurance

The quality gate prevents *preventable* failures, not all failures. Consider a production out-of-memory crash caused by concurrent users growing beyond the original capacity plan: the environment *was* load-tested before promotion against its declared `loadTestConfig` thresholds (for example, 50 concurrent users, P95 $\leq$ 200 ms, 99.5% success) and passed; the crash occurred because *actual* load exceeded the *configured* threshold. This is by design—the gate validates against declared business requirements, not unbounded traffic—and the correct response is to update the thresholds to reflect the new reality, a one-line YAML change in the custom resource that takes effect on the next promotion cycle.

This pattern—declare thresholds, gate against them, update when reality diverges—transforms load testing from a periodic manual activity into a continuous, declarative contract versioned in Git. A natural extension (Section 11.5) is a controller that watches observability metrics and adjusts `loadTestConfig` thresholds from actual production traffic, closing the loop between observability and quality gating.

8.2.7 Operational Edge Cases

Seven edge-case scenarios were tested against the proof-of-concept deployment to verify system behaviour under non-ideal conditions. Table 8.3 summarises the results.

Six of the seven edge cases passed; the seventh is recorded as *not reproducible* rather than failed: ArgoCD's `selfHeal` policy restored the scaled-to-zero deployment faster than the EventSource–Sensor–Workflow pipeline could trigger the health check. This is a positive finding for operational resilience—`selfHeal` and the quality gate reinforce each other—but the 5-minute timeout path is verified only indirectly, observed when a staging gate genuinely failed with no deployment in the target namespace.

Table 8.3: Operational edge-case test results on `private.novelcore.org`. ✓ = scenario reproduced and behaved as expected; n/r = scenario not reproducible under the deployed configuration (not a failure).

Scenario	**Observed Behaviour**	**Pass**
Webhook redelivery	Redelivered webhook was accepted (HTTP 200) and processed; new workflow triggered.	✓
Workflow re-trigger	Closing and reopening a PR re-triggered the gate workflow via EventSource/Sensor. New workflow created and completed.	✓
Health-check timeout	ArgoCD `selfHeal` restored the scaled-to-zero deployment before the health check could detect unavailability. Timeout scenario not reproducible with selfHeal enabled.	n/r
Manual approval	Production PR remained open despite passing all gates (`autoMerge: false`). CommitStatus showed `success`; manual merge required.	✓
Branch protection	XSDLC-provisioned BranchProtection rules enforced status checks (`staging-gate`, `prod-gate`) on all 5 API repositories. 15 BranchProtection CRDs all Synced/Ready.	✓
CommitStatus lifecycle	32 CommitStatus CRDs observed across 5 APIs. Phase distribution: 18 failure, 13 success, 1 pending. Each commit SHA received its own CRD.	✓
ArgoCD sync after merge	Both `perfect-api-staging` and `slow-api-staging` reached Synced/Healthy state with running pods after staging PR merge.	✓

The multi-API evaluation also exposed a reporting inconsistency in the `load-only` CLI command: the `slow-api` production load test produced a JSON report with `"exit_code": 0` despite `"status": "failed"`. The process exit code was correct (exit code 1), so the workflow detected the failure regardless; the bug was fixed by computing `result.Status` from the `FailedChecks` count before serialising the report. This is a self-testing discovery: the framework's own gate infrastructure, running the framework's own CLI, exposed an inconsistency that unit tests had not covered.

8.2.8 CI/CD Integration Metrics

The PoC's final output is a set of metrics quantifying DriveBy's integration footprint in the delivery pipeline. These are *technical-integration* metrics (footprint and latency of the gate machinery itself), not delivery-performance metrics in the DORA sense: deployment frequency, lead time, change failure rate, and time to recovery require a production change stream far longer than this PoC window, and collecting them is recorded as future operational work. Table 8.4 reports what the PoC measures directly.

Table 8.4: CI/CD integration metrics for the DriveBy framework. Latency rows report values observed during the PoC; size rows are properties of the released artefacts.

Metric	**Value**	**Notes**
CLI binary size	∼15 MB	Cross-compiled Go binary
Docker image size	∼25 MB	Distroless base image
Startup time	<1 s	No JVM, no runtime dependencies; consistently sub-second in PoC gate executions
Validation time (strict)	2–5 s	Measured per run; depends on specification size
XSDLC provisioning time	∼15 s	From CR apply to all resources ready
End-to-end gate latency	40–70 s typical	Webhook to commit status posted; median 69 s across the 44 PoC workflow runs, with a ∼334 s tail for runs that waited out the 5-minute health-check timeout
Resources per XSDLC CR	∼44–47	For a 3-environment pipeline
Helm chart size	<50 KB	OCI-distributed

The two size rows are justified as follows: every gate execution pulls the DriveBy container image into an ephemeral workflow pod, so binary and image size directly bound the cold-start cost of each gate run. They are footprint properties of the automated approach with no manual-process counterpart.

For the metrics that *do* have a manual counterpart, Table 8.5 contrasts the measured automated values with the manual equivalent each activity replaces. No manual control arm was run, so the manual column carries engineering estimates (consistent with the ClickOps analysis in Section 8.2.4); the comparison is indicative of magnitude, not a controlled result.

Table 8.5: Automated SDT pipeline versus the manual equivalent it replaces. Automated values are measured in the PoC; manual values are engineering estimates (no manual control arm was run).

Activity	**Automated (measured)**	**Manual equivalent (estimate)**
Pipeline provisioning	∼15 s from a single ∼35-line CR	Hours of ClickOps across the ArgoCD UI, GitHub settings, and `kubectl`; error-prone and non-reproducible
Promotion-gate validation	2–5 s per run, zero test authorship	Manual contract review per release, plus authoring and maintaining the check list
Load-test execution	Declarative `loadTestConfig`; 15 s test embedded in the gate run	Scripting, executing, and interpreting an ad-hoc load test per promotion
Result interpretation	Machine-readable verdict posted as a CommitStatus within the gate latency	Human reading and judging a test report per promotion
Reproducibility	Versioned CR; identical re-runs (Axiom of Determinism)	Dependent on the individual operator; not version-controlled

8.3 Controlled Evaluation: non-critical-api

Naming note. The reference API is referred to throughout the source code, deployment manifests, and historical artefacts as `perfect-api`; it is renamed here to `non-critical-api` because the previous name was misleading. Under the severity assignment of Chapter 3, the new name reflects the empirical property of the reference: it carries *no critical-severity defects*, passing every implemented principle (P001–P005, P008, P009) in every mode in which it is evaluated, with no critical or warning findings. Source-code identifiers retain the historical name for stability; the rename is a thesis-level naming decision to be propagated to source in a follow-up release.

The controlled evaluation uses this reference API as a positive control for critical-severity principles, then systematically injects defects to verify detection accuracy across the full principle space.

8.3.1 Reference API Design

The `non-critical-api` is a small *product management* API: a Python FastAPI application with a hand-crafted OpenAPI specification engineered to satisfy every implemented principle in DriveBy's strict mode. It defines eight active operations across three resource groups—five CRUD endpoints for `/products`, a `POST /tasks` endpoint, and two utility endpoints (health, echo)—plus a deprecated `GET /legacy/products` endpoint that is excluded from functional testing. The specification includes a complete error-response surface (400, 401, 403, 404, 500) with consistent schemas, reusable `components.responses` entries, typed parameters with full length and range constraints, OAuth2 security with scopes referenced at the operation level, semantic versioning with migration documentation, and response examples for all 2xx responses. The reference is clean at the runtime layer as well as the static layer: it passes P006 functional testing and clears its P007 load-test thresholds in the operational pipeline (Section 8.4.4), qualifying it as the base from which behavioural faults can be injected. The API is deployed to the `perfect-api-dev` namespace (the historical artefact name) on the target cluster and serves as the validation target for the controlled and operational arms.

8.3.2 Baseline Validation

At baseline, `non-critical-api` passes every implemented principle in every evaluated mode (Table 8.6); the recorded JSON report

(`results/local/perfect-api-validate-strict.json`) carries top-level `"status": "passed"` with no critical or warning findings.

Table 8.6: Baseline validation results for `non-critical-api` across validation modes, reproduced from `results/local/perfect-api-validate-{strict,test-ready}.json`; the minimal-mode row follows directly from the mode definition (P001 only). ✓ = principle fully passes in this mode; "—" = principle is not evaluated in this mode.

Mode	**P001 (C)**	**P002 (C)**	**P003 (C)**	**P004 (C)**	**P005 (C)**	**P008 (W)**	**P009 (W)**	**Result**
Minimal	✓	—	—	—	—	—	—	1/1 passed
Test-ready	✓	✓	✓	✓	—	—	✓	5/5 passed
Strict	✓	✓	✓	✓	✓	✓	—	6/6 passed

"—" indicates the principle is not evaluated in that mode. Severity: **C** = critical (blocks delivery), **W** = warning (surfaced but non-blocking).

This baseline establishes the zero-false-positive property: a hand-crafted specification engineered to satisfy every principle produces no spurious findings in any mode.

8.3.3 Defect Injection and Detection

To verify detection accuracy, seven defects are injected into the `non-critical-api` specification, one targeting each implemented principle and designed to violate it alone (Table 8.7):

All seven injected defects are detected with zero false negatives and zero false positives (no unrelated principles are affected by a single-principle defect). The detection is deterministic: repeated runs against the same defective specification produced identical results, confirming the Axiom of Determinism.

8.3.4 Per-Check Extension and Combination Robustness

The seven-defect protocol above tests one defect per principle; it shows neither full *check-level* coverage within each principle nor behaviour under *multiple simultaneous* defects (a realistic scenario in specs that drift over time). An extended protocol therefore injects defects at the level of individual checks and of cross-principle combinations.

Table 8.7: Defect injection matrix: each injected defect targets one principle.

Target	Defect	Description	Detected
P001	Duplicate operationId	Two endpoints share the same `operationId`	✓
P002	Missing descriptions	Remove all operation summaries and descriptions	✓
P003	Missing error responses	Remove all 4xx/5xx response definitions	✓
P004	Untyped schemas	Remove type specifications from request body schemas	✓
P005	Missing security	Remove `components.securitySchemes` entirely	✓
P008	Invalid version	Change `info.version` to "latest" (non-semver)	✓
P009	Missing examples	Remove all parameter examples and schema types	✓

A reproducible Python harness (`tools/per-check-defect-injection.py`) generates fifteen check-level mutations of the `non-critical-api` baseline—two to three per implemented principle, each targeting a single named check—and runs DriveBy in strict mode against each (Table 8.8). A second pass generates four randomised mutation combinations (three pairs and one triple spanning different principles) and verifies that every targeted principle is flagged simultaneously. One pair could not be evaluated: its mutated specification triggers a nil-pointer panic inside the upstream `kin-openapi` parser, aborting the validator before any principle runs—an upstream library defect, recorded as such rather than as a detection miss. All artefacts are persisted under `results/per-check-injection/`. The protocol samples the check space rather than exhausting it; complete coverage is recorded as future evaluation work.

Combination robustness. In every evaluable combination, all targeted principles were detected as failing in the same run—confirming that the principle checkers operate independently and that simultaneous defects do not mask one another (Table 8.9).

Interpretation of misses. The two undetected single-check mutations have different natures. The `p005-strip-op-security` miss is an injection artefact rather than a framework limitation: the removed per-operation `security` entries are redundant under the intact global `security` requirement, so the mutated specification is still fully protected and there is no real defect to report. The `p009-strip-param-examples` miss, by contrast, is a genuine

Table 8.8: Per-check defect-injection results for `non-critical-api`. Detection means the targeted principle either (a) transitions from passing to failing, or (b) reports a new failed check that was passing in the baseline.

Mutation	**Target**	**Outcome**
duplicate operationId	P001	detected
invalid HTTP method	P001	detected
strip operation/schema descriptions	P002	detected (with P008 collateral)
strip request/response examples	P002	detected
strip contact and license	P002	detected
strip 4xx/5xx response definitions	P003	detected
strip `components.responses`	P003	detected
remove `type` from request bodies	P004	detected
strip schema length/range constraints	P004	loader-rejected (parse error)
delete `components.securitySchemes`	P005	detected
delete top-level `security`	P005	detected
delete per-operation `security`	P005	*missed*
non-semver `info.version`	P008	detected
strip versioning-strategy keywords	P008	detected
strip parameter-level examples	P009	*missed*
Single-check detection rate: 12/15 (80%); collateral failures: 1.		

Table 8.9: Combination defect-injection results: simultaneous mutations on the `non-critical-api` baseline.

Combination kind	**Targeted principles**	**Outcome**
pair	P001 + P002	all detected
pair	P001 + P003	all detected
triple	P002 + P003 + P005	all detected (P008 also flagged as collateral)
Combination detection rate: 3/3 evaluable (all targets caught); 1 pair unevaluable (upstream parser panic).		

framework limitation: P009's example-presence check operates on schema-level rather than parameter-level examples, so the targeted defect bypasses the check by definition. The genuine miss is recorded in the limitations discussion of Chapter 10 and as a candidate strengthening in Chapter 11; the injection artefact is recorded as a defect-generator refinement for future rounds.

Interpretation of the collateral. The `p002-strip-descriptions` mutation also caused P008 to fail, because P008's "versioning strategy is documented" check looks for keywords in `info.description`—which the mutator stripped along with operation descriptions. This is principled cross-talk (a real overlap between checks), not a false positive: a spec stripped of its `info.description` genuinely does not document its versioning strategy.

Interpretation of the parse error. The `p004-strip-constraints` mutation produced a specification the OpenAPI loader rejected before any principle checker ran—a P001-level catch at the I/O boundary, not a P004 miss, since some constraint annotations are not optional in OpenAPI 3.1. It demonstrates a useful property of the layered architecture: structural invalidity short-circuits semantic validation.

Determinism check. The harness is invoked with a fixed random seed (42) and re-running it produces bit-identical `summary.json` output, confirming the Axiom of Determinism at the experimental scale.

Conclusions from the per-check extension. Fourteen of the fifteen single-check mutations are accounted for (twelve detected, one neutralised by the loader's structural validation, one an injection artefact with no real defect), every evaluable multi-defect combination was fully detected, and repeated runs are bit-identical. The remaining limitations—P009's parameter-level blind spot, the sampled rather than exhaustive check space, and the combination unevaluable due to an upstream parser defect—are carried into Chapters 10 and 11.

8.4 Multi-API Controlled Evaluation

The single-API controlled evaluation (Section 8.3) established detection accuracy through defect injection. This section extends the evaluation to a five-API suite, where each API embodies a distinct defect class and is validated through the full XSDLC quality gate pipeline deployed on `private.novelcore.org`.

8.4.1 Evaluation API Suite Design

Five purpose-built FastAPI applications, each derived from the `non-critical-api` reference by surgically removing or altering one quality dimension while preserving all others, exercise the full SDT principle space. The design intent of each API is given in Table 8.10; the empirically measured outcome follows in Table 8.11, reproduced verbatim from the JSON reports in `results/local/` so that it cannot drift from the data.

Table 8.10: Evaluation API suite: each API targets a distinct defect class.

API	Target Defect Class	Expected Failure	OpenAPI
non-critical-api	Positive control (no by-construction defects)	none	3.0.3
bad-docs-api	Stripped documentation & error metadata	P002, P003, P004 (C); P008 (W)	3.0.3
no-auth-api	Missing authentication schemes	P005 (critical)	3.0.3
slow-api	Performance degradation (500 ms delay)	P007 (load test, C)	3.0.3
broken-api	Spec–implementation mismatch	P006 (functional, C)	3.0.3

`bad-docs-api` strips descriptive content: operation summaries and descriptions, examples, and error-response metadata. `no-auth-api` removes the security configuration (`components.securitySchemes` and the security requirements referencing it). The two behavioural-defect APIs leave the specification byte-identical to the reference and mutate only the implementation: `slow-api` inserts an artificial 500 ms delay (`asyncio.sleep(0.5)`) into every request handler, so every endpoint responds correctly but slowly; `broken-api` mutates response status codes so the implementation diverges from the contract (`GET /products` returns HTTP 418 instead of the documented 200, `POST /products` returns 200 instead of the documented 201).

The suite was designed to verify three properties: (1) each defect class is detected by its corresponding principle, (2) the severity model correctly distinguishes blocking from non-blocking failures, and (3) the layered gate architecture (staging then production) provides defense in depth.

Two observations follow from the measured matrix. First, each API fails *exactly its targeted defect class and nothing else.* The documentation-class strip of `bad-docs-api` legitimately cascades across the documentation-adjacent principles P002/P003/P004/P008, because a single strip of descriptive content

Table 8.11: Empirically measured strict-mode outcomes per API (source: `results/local/<api>-validate-strict.json`). **P** = principle fully passed; **F** = at least one check failed. P006 and P007 are runtime principles not exercised by static validation; their column reports the runtime-layer outcome: "runtime pass" = reached the runtime layer and passed, "runtime fail" = reached the runtime layer and failed, "–" = blocked statically before the runtime layer was reached.

API	**P001 (C)**	**P002 (C)**	**P003 (C)**	**P004 (C)**	**P005 (C)**	**P008 (W)**	**P006/P007 (runtime)**
non-critical-api	P	P	P	P	P	P	runtime pass
bad-docs-api	P	F	F	F	P	F	–
no-auth-api	P	P	P	P	F	P	–
slow-api	P	P	P	P	P	P	runtime fail (P007)
broken-api	P	P	P	P	P	P	runtime fail (P006)

removes evidence that all four independently check; the security strip of `no-auth-api` surfaces as exactly one violation (P005). Second, the runtime-defect APIs pass all six static principles: their specifications are byte-identical to the reference, so their defects are invisible to static analysis and are caught by the runtime layer (P006, P007) instead, completing the layered defence with clean per-layer discrimination.

The suite *does not* include a critical-P001 example (every spec parses cleanly) but does include critical-P005 and critical-P006 examples. The omission is deliberate: a critical-P001 specimen is trivial to construct but would exercise only the loader's parse stage rather than a principle checker, and that loader-rejection path is already characterised by the constraint-strip mutation of Section 8.3.4.

8.4.2 Local Validation Results

Each API was first validated locally using the DriveBy CLI in strict mode; Table 8.12 presents the per-principle pass/fail results.

The static gate blocks exactly the two specification-defect APIs (Table 8.12); the runtime-defect APIs score a clean 6/6 statically and are caught exclusively by the runtime layer—`broken-api` on P006 (HTTP 418 instead of the documented 200) and `slow-api` on P007 (latency above the 200 ms threshold). Each defect class is detected at exactly the layer designed to catch it, and none escapes detection.

Table 8.12: Local validation results in strict mode (6 principles). ✓ = pass, ✗ = fail. Severity: **C** = critical, **W** = warning. Critical-failure counts reflect the severity assignment in which P001–P005 are all critical (Section 3.4).

API	P001 (C)	P002 (C)	P003 (C)	P004 (C)	P005 (C)	P008 (W)	Passed	Critical Failures
non-critical-api	✓	✓	✓	✓	✓	✓	6/6	0
bad-docs-api	✓	✗	✗	✗	✓	✗	2/6	3
no-auth-api	✓	✓	✓	✓	✗	✓	5/6	1
slow-api	✓	✓	✓	✓	✓	✓	6/6	0
broken-api	✓	✓	✓	✓	✓	✓	6/6	0

8.4.3 Staging Gate Results

Each API was deployed through the XSDLC pipeline, triggering staging gate workflows via webhook-driven Argo Events sensors; gate composition follows the PoC configuration of Table 8.1. The mode asymmetry visible in the table—`test-ready` for the reference, `strict` for the defect APIs—is the deliberate design choice of Section 8.2.2, exercising the mode-selection mechanism end-to-end; because the reference passes both modes (Table 8.6), it does not affect the cross-API comparison. Table 8.13 lists each gate's checks alongside the observed outcome.

Three APIs were blocked at the staging gate (Table 8.13); notably, `broken-api` passed static validation cleanly—its specification is identical to the reference—and was caught at the runtime layer on P006, demonstrating that runtime validation catches behavioural drift even when the specification carries no static defects. `slow-api`, by contrast, *passed* its staging gate: its specification is clean, its endpoints answer functional testing's single requests correctly, and its 500 ms latency defect—invisible to the staging gate's checks, which include no load test—was caught at the production gate instead (Section 8.4.4).

8.4.4 Production Gate Results

Only two APIs reached the production gate; three were blocked at staging and never promoted (Table 8.14).

The `non-critical-api` load test ran 150 requests over 15 seconds with 10 concurrent users, recording a P50 latency of 3.7 ms, P95 of 4.9 ms, and P99 of 5.8 ms—well within the gate's 2 s P95 threshold (Table 8.1). The

Table 8.13: Staging gate results for all five evaluation APIs.

API	Gate Checks	Result	Blocking Failure
non-critical-api	validate-only (test-ready) + functional-test	✓ PASS	—
slow-api	validate-only (strict) + functional-test	✓ PASS	—
bad-docs-api	validate-only (strict)	✗ FAIL	P002/P003/P004 Documentation (critical)
no-auth-api	validate-only (strict)	✗ FAIL	P005 Security (critical)
broken-api	validate-only (strict) + functional-test	✗ FAIL	P006 Functional (critical)

Table 8.14: Production gate results. "Not reached" indicates the staging gate failed; the production gates configured for those APIs (Table 8.1) were therefore never exercised.

API	Gate Checks	Result	Blocking Failure
non-critical-api	load-test (10 users, 15 s)	✓ PASS	—
slow-api	validate-only (strict) + load-test	✗ FAIL	P007 P95 latency 505 ms > 200 ms
bad-docs-api	not exercised	Not reached	Blocked at staging (P002/P003/P004)
no-auth-api	not exercised	Not reached	Blocked at staging (P005)
broken-api	not exercised	Not reached	Blocked at staging (P006)

production promotion PR remained open (`autoMerge: false`), requiring manual approval before merging.

The `slow-api` passed static validation at the production gate (identical to staging: a clean 6/6), but the load test recorded a P95 latency of 505 ms—more than double the 200 ms target. The 500 ms artificial delay, invisible to single-request functional testing, manifested clearly under concurrent load.

Of the five APIs, only `non-critical-api` (20%) passed every gate from development through staging to the production boundary; its production promotion PR was gate-approved and awaited only the deliberate manual merge required by `autoMerge: false`.

8.4.5 Layered Validation Defence

The five-API evaluation revealed a three-layer defence model, where each layer caught a distinct class of defect that the preceding layers could not detect (Table 8.15):

Table 8.15: Layered validation defence: each layer catches defects invisible to prior layers.

Layer	Check Type	Defect Class	APIs Caught
1	Static validation (P001–P005, P008)	Specification defects	bad-docs-api, no-auth-api
2	Functional testing (P006)	Implementation defects	broken-api
3	Load testing (P007)	Performance defects	slow-api

Each layer was necessary: static validation cannot detect `broken-api`'s status-code drift or `slow-api`'s latency profile (inherently runtime properties), and functional testing alone would have passed `slow-api`, whose endpoints answer single requests correctly. Only the combination of all three layers provided the discrimination required to identify all four defect classes.

8.4.6 Severity Model in Practice

The severity assignment (defined in Chapter 3) reflects SDT's core design principle: *validation gates functional testing.* Seven of the nine principles are Critical—the static P001–P005 plus the runtime P006 and P007, whose failures block within their respective gate types; a declared performance threshold is a contract, so a P007 breach always blocks promotion exactly as a

static critical failure does—there is no advisory mode for a violated SLA. The 5/7 static-critical ratio reflects the gatekeeping function of static validation:

- P002 (Documentation) is critical because functional test generators require examples and operation descriptions to produce valid request payloads; without them, test data is generic and rejected by API validation.
- P003 (Error Handling) is critical because negative test assertions require documented error schemas; without them, tests cannot verify error response shapes.
- P004 (Schema Definitions) is critical because type-less schemas force test generators to fall back on untyped placeholder values.
- As a result, the specification-defect APIs are blocked at static validation and the behavioural-defect APIs at the runtime stage—the correct behaviour: an incomplete specification should not proceed to functional testing, while a complete specification with a defective implementation should fail exactly there.
- The only warning-severity principles (P008, P009) represent quality dimensions that degrade gracefully—versioning metadata and response-example advisories—without preventing test execution.

8.5 Large-Scale Evaluation: APIs.guru

The large-scale evaluation applies SDT validation to real-world API specifications from the APIs.guru registry [6], assessing how existing APIs perform against SDT's quality standards.

8.5.1 Dataset and Protocol

The evaluation harvested API specifications from the APIs.guru registry using `tools/harvest-openapi-apisguru.py`. The dataset comprises **70 specifications in total**: 50 OpenAPI 3.x (26 at 3.0.x, 24 at 3.1.0) and 20 Swagger 2.0, drawn from major API providers (Stripe, Twilio, GitHub, AWS) as well as smaller independent APIs. Results are reported per version family (Sections 8.5.3 and 8.5.4) because the two arms exercise different specification adapters and the version-agnosticity claim requires per-adapter evidence; the 2.0 arm reproduces the 3.x pattern, so the split changes no conclusion. Each specification is validated in both *minimal* and *strict* modes

using `tools/run-openapi-batch.sh`. Only the static principles of strict mode (P001–P005, P008) are evaluated: P006 (Functional Testing) and P007 (Performance Testing) are excluded because the evaluation lacks authenticated access to the live API endpoints, and P009 (Test Readiness)—although static—is excluded because it belongs to the test-ready mode set, not the strict mode applied uniformly across the sample (Table 8.6), and its advisory checks presuppose the downstream functional testing this protocol does not perform.

The evaluation was conducted through an agent-driven workflow (Chapter 9): the AI agent developed the harvesting and batch-testing tooling, executed the runs, parsed the structured JSON reports, and drafted the analysis; the human developer reviewed the interpretation and verified the numbers against raw output. The workflow itself validates a key SDT claim: the machine-readable output required by the Axiom of Observability is directly consumable by automated actors, including AI agents performing empirical analysis.

8.5.2 Baseline: Swagger Petstore v3

The Swagger Petstore v3—the canonical OpenAPI example maintained by the OpenAPI Initiative—serves as a reference data point. In strict mode, Petstore scores **1/6**: only P001 (OpenAPI Specification Compliance) passes. The five failing principles reveal systematic documentation quality gaps:

- **P002** (Documentation Quality): Missing operation descriptions, parameter descriptions, and response examples across most endpoints.
- **P003** (Error Handling): No 5xx error responses defined; error response schemas lack structured fields.
- **P004** (Schema Definitions): String parameters lack length constraints; numeric fields lack range constraints.
- **P005** (Security): Security schemes defined, but their scopes (for `oauth2`/`openIdConnect` schemes) are not referenced in operation-level security requirements.
- **P008** (Versioning): Version field present and follows semver, but no versioning strategy, deprecation notices, or migration guides documented.

This result is significant: *even the canonical example specification maintained by the OpenAPI Initiative fails SDT's strict validation*, supporting the thesis claim that existing specifications systematically neglect exactly the gaps SDT is designed to detect.

8.5.3 Per-Principle Pass Rates

This subsection reports the OpenAPI 3.x arm of the dataset (raw artefacts under `results/apisguru-rerun/`): DriveBy executed in strict mode against the **50 OpenAPI 3.x specifications**. All 50 validated successfully; no timeouts or harness errors occurred.

Per-principle pass rates are reported in Table 8.16 as **exact counts and percentages**; bucketing (**high** $\geq 80\%$, **medium** 30–79%, **low** $< 30\%$) is retained only as a readability aid. The thresholds are pragmatic rather than theoretically derived, and the buckets carry no analytical weight—all analysis below operates on the exact counts.

Table 8.16: Per-principle pass rates across the 50-API APIs.guru sample (strict mode). Bucket thresholds: high $\geq 80\%$, medium 30–79%, low $< 30\%$.

ID	Principle	Pass	Fail	Pass rate	Bucket	Severity
P001	OpenAPI Compliance	23	27	46.0%	medium	critical
P002	Documentation Quality	0	50	0.0%	low	critical
P003	Error Handling	0	50	0.0%	low	critical
P004	Schema Definitions	6	44	12.0%	low	critical
P005	Security Standards	13	37	26.0%	low	critical
P008	Versioning Strategy	0	50	0.0%	low	warning

The headline pattern: **three of the six implemented principles (P002, P003, P008) record a pass rate of zero**—no public API in the sample fully satisfies them. Only P001 (OpenAPI structural compliance) reaches the medium bucket at 46%, and even that is lower than expected—structural compliance with the published OpenAPI specification is not universal in the wild. The P005 result deserves a separate reading: the 13 passing APIs declare their security schemes *and* reference them at the operation level; the 37 failing APIs typically declare schemes but do not require them. This is more often a *test-orientation and usage-documentation gap*—neither a test generator nor an API consumer can tell from the specification which calls require credentials—than a *security gap* per se, and it is exactly the failure mode the SDT principle is designed to surface.

Score distribution. Aggregating across principles gives the distribution in Table 8.17.

84% of the sampled public APIs score 1/6 or worse; **no API in the sample passed even half** of the strict-mode principles. The entire sampled population fails the same handful of completeness principles; combined with

Table 8.17: Strict-mode score distribution across the 50-API sample. Score = number of principles fully passing (out of six evaluated).

Score	Count	%
0/6	16	32.0%
1/6	26	52.0%
2/6	8	16.0%
3/6	0	0.0%
4/6	0	0.0%
5/6	0	0.0%
6/6	0	0.0%

the zero-false-positive controlled baseline—which affirms that a specification *can* carry all the information the principles demand—this supports the thesis position that a systemic specification-quality gap exists which automated, deterministic enforcement is needed to close.

The gap between P001 (46%) and P002–P004 (zero or near-zero) confirms that existing tooling validates specification *form* (structural compliance) but not *substance* (documentation completeness, schema precision)—and the P001 number itself warrants a stronger reading: at 46%, the *majority* of sampled public specifications fail validation against the published OpenAPI grammar and are not reliably machine-processable at all.

The five-API controlled evaluation (Section 8.4) provides the contrasting baseline, visualised in Figure 8.2: specifications engineered for completeness sit at 80–100% per principle (each single failure tracing directly to the deliberate defect injection) while the public population sits at 0–46%, confirming that the gap is a quality property of the specifications, not an artefact of the validator.

8.5.4 Swagger 2.0 Coverage

Swagger 2.0 specifications still represent a substantial fraction of public APIs in production use, so this section reports a separate 20-API evaluation arm sampled from the Swagger 2.0 portion of the APIs.guru registry.

Method. The harvester's (`tools/harvest-openapi-apisguru.py`) `2.0` version selector scans every version listed for an APIs.guru entry, since the registry prefers 3.x as each entry's *preferred* version; a per-provider cap of two

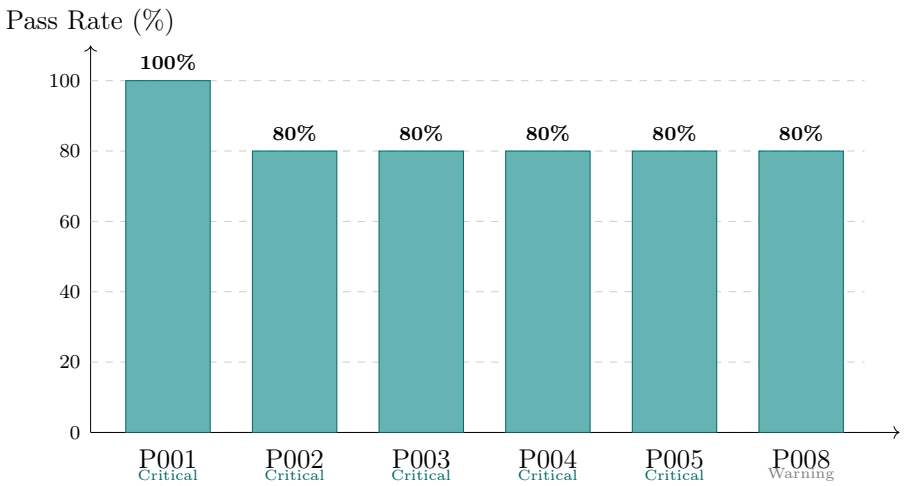

Figure 8.2: Per-principle pass rates across the five evaluation APIs (strict mode). P001 achieves universal compliance; each remaining principle fails for exactly one API, reflecting the targeted defect injection (`bad-docs-api` for P002–P004/P008, `no-auth-api` for P005).

specifications ensures 20 distinct providers rather than 20 sub-APIs of a single vendor (`amadeus.com` alone publishes more than 30 Swagger 2.0 sub-APIs). DriveBy was run in strict mode against each harvested URL, producing per-principle reports under `results/swagger2-evaluation/reports/`.

Operational outcome. All 20 harvested specifications were parsed and validated successfully—no parse errors, no adapter crashes, no spec-conversion failures. The same invocation that drives the 3.x arm drives the 2.0 arm with no flag changes; version detection at load time routes the document to the appropriate adapter, after which the same checker registry is exercised (Chapter 5). This confirms that DriveBy is genuinely version-agnostic in operation, not merely in claim.

Per-principle results. Table 8.18 reports pass rates across the 20 Swagger 2.0 specifications.

Score distribution. 16 of 20 specifications (**80%**) scored 1/6, four scored 2/6, and none scored higher—**no Swagger 2.0 specification passed even half** of the strict-mode principles, a pattern strongly consistent with the 84% rate at 1/6-or-worse in the 3.x arm. The systemic specification-quality gap that motivates SDT is therefore not an artefact of the OpenAPI 3.x

Table 8.18: Per-principle pass rates on the 20-API Swagger 2.0 sample (strict mode).

Principle	**Severity**	**Pass**	**Rate**
P001 — OpenAPI Compliance	Critical	20/20	100%
P002 — Documentation Quality	Critical	0/20	0%
P003 — Error Handling	Critical	0/20	0%
P004 — Schema Definitions	Critical	1/20	5%
P005 — Security Standards	Critical	3/20	15%
P008 — Versioning Strategy	Warning	0/20	0%

population: it reproduces in 2.0 with statistically indistinguishable severity. One per-principle result diverges across the arms: P001 passes universally here (every harvested spec is structurally valid 2.0), against 46% in the 3.x arm. A plausible explanation is toolchain maturity: specifications still published in 2.0 form have survived a decade of strict parser ecosystems, whereas the 3.x sample includes newer 3.1.0 documents whose structural conventions are less uniformly enforced. P002, P003, and P008, by contrast, fail across the entire sample in both arms.

Summary. DriveBy validates real-world 2.0 specifications end-to-end with the same per-principle output structure as 3.x—so downstream consumers such as the agent-feedback loop (Section 8.6) work without modification—and the 3.x empirical pattern reproduces in 2.0; raw artefacts are version-controlled under `results/swagger2-evaluation/`.

8.6 SDT Feedback as Input to Agent-Driven Specification Improvement

This section reports a small but empirically anchored experiment testing a claim the framework has so far made on theoretical grounds: that the structured output of a DriveBy validation run is, by itself, sufficient to drive autonomous specification improvement by an AI agent. The result is an empirical proof at the unit-of-feedback scale.

8.6.1 Experimental Design

The experiment uses the canonical *Swagger Petstore* OpenAPI specification (`petstore3.swagger.io`, OpenAPI 3.0.x, 13 paths, 19 operations), previously reported in Section 8.5 as scoring 1/6 in strict mode. The choice of a public, third-party specification (rather than one of the controlled-evaluation APIs) is deliberate: the experiment tests agent-driven correction on a specification the agent has not previously seen as part of the thesis pipeline.

The experimental treatment proceeds in three steps:

1. **Baseline measurement.** The unmodified Petstore spec is fetched directly from `petstore3.swagger.io` and validated by DriveBy in strict mode (P001–P005, P008). The structured JSON report is persisted.

2. **Agent revision (sole-feedback constraint).** An AI coding agent (Claude Code, Sonnet 4.6) is instructed to revise the specification with two strict experimental constraints: (i) the agent may read *only* two files—the baseline OpenAPI spec and the DriveBy validation report; and (ii) the agent may not access the web, the DriveBy CLI source, the thesis chapters, the project documentation, or any other API specification. The validation report is therefore the agent's sole feedback signal. The agent writes a single revised specification and an audit trail listing every field it added or changed, grouped by the principle it targeted.[1]

3. **Post-revision measurement.** The revised specification is validated by DriveBy in the same strict mode. The score delta is computed per principle.

All experimental artefacts—baseline spec and report, audit trail, revised spec, post-revision report—are saved under `results/sdt-feedback/` as the reproducible evidence base for this section.

8.6.2 Results

Table 8.19 reports the per-principle score delta; the score is the count of the six strict-mode principles whose checks fully pass.

The headline result is a **three-fold improvement (1/6 → 4/6)** in a single agent-revision pass with the DriveBy validation report as the sole feedback channel. Four of the five baseline failures (P002, P004, P005,

[1]The full instruction and the audit trail are archived at `results/sdt-feedback/petstore-revised/agent-changes.md`, alongside the baseline and revised specifications and all validation reports.

Table 8.19: SDT feedback experiment: per-principle outcomes for the Swagger Petstore specification, before and after a single agent-revision pass driven solely by the DriveBy validation report. ✓ = pass, ✗ = fail.

Principle	**Baseline**	**Revised**	**Δ**	**Notes**
P001 Compliance	✓	✗	−1	Regressed: stale XML example on `Pet` request body
P002 Documentation	✗	✓	+1	Operation/schema descriptions, examples added
P003 Errors	✗	✗	0	Partial: 4xx/5xx wired, but `components.responses` reuse missing
P004 Schema	✗	✓	+1	Length, format, min/max constraints added
P005 Security	✗	✓	+1	Top-level security + per-operation security wired
P008 Versioning	✗	✓	+1	Semver, breaking-changes, migration narrative added
Total passed	**1/6**	**4/6**	**+3**	Net of one regression and one partial fix

P008) were fully resolved; one was partially addressed (P003: the agent wired 4xx/5xx responses to all 19 operations but did not introduce the `components.responses` reuse pattern that DriveBy's check requires); and one previously-passing principle (P001) regressed because the agent left a stale XML-formatted example on the `Pet` request body.

8.6.3 Interpretation

Three observations follow.

First, the experiment confirms the theoretical claim made in Chapter 9: *SDT's structured output is sufficient to drive measurable, autonomous specification improvement.* The agent received no instructions about what specific edits to make—only the JSON report—and produced a spec the validator scored substantially higher, empirically grounding the framework's positioning as *feedback infrastructure for agent-driven development*, not merely a static linter.

Second, the experiment also confirms that single-pass agent revision is *not* a substitute for human review. The P001 regression illustrates a class of failure mode peculiar to LLM editors: when the agent enriches one part of the spec, it can introduce inconsistencies elsewhere that a less-thorough validator would miss. The post-revision DriveBy report is, however, exactly the right input to a second agent pass, and the four-of-five conversion in a single pass is encouraging evidence that two or three iterations would converge to a fully-passing spec.

Third, the P003 partial result is itself diagnostic: the agent wired the 4xx/5xx responses but inlined them rather than referencing the *shared* component-level error responses P003 demands, and the follow-up DriveBy run flagged exactly this—the validate–remediate–revalidate loop functioning correctly, each iteration's report carrying strictly more specific information than the last.

A further design limit deserves explicit statement: revising from the report alone, the agent cannot know which operations *should* be protected or which errors the implementation *actually* produces, so report-only revision risks a specification that diverges from the implementation; P006 functional testing catches such divergence only to the extent its coverage exercises the affected paths. Source-aware revision—giving the agent read access to the implementation alongside the report—is future work, as is the variant of supplying the agent with the principle *definitions* in addition to the report, which might converge in fewer rounds.

The experiment scope is deliberately narrow—one specification, one agent, one revision pass—and the section's claim is correspondingly bounded: SDT

feedback drives measurable correction *at the unit-of-feedback scale*. Broader claims (e.g. that iterative agent loops always converge, or that this transfers across specifications) require the multi-spec, multi-iteration follow-on study that is recorded as future work in Chapter 11.

8.7 Summary

Across its four arms the evaluation establishes: check-level detection accuracy with deterministic re-runs and exactly one genuine framework miss (Section 8.3.4); clean per-layer discrimination of all four injected defect classes in the five-API pipeline, with zero misclassifications and only the reference reaching production (Section 8.4); a quantified, population-wide specification-quality gap that reproduces across the OpenAPI 3.x and Swagger 2.0 arms (Section 8.5); single-pass agent-driven improvement of Petstore from 1/6 to 4/6 with the validation report as sole feedback (Section 8.6); and an operational PoC running five concurrent `XSDLC` pipelines with sub-minute typical gate latency (Sections 8.2 and 8.2.8). These results evidence RQ1 (detection accuracy and generalisation), RQ2 (derivation and agent-actionability), and RQ3 (GitOps integration), while RQ4 receives its primary evidence in Chapter 6 with partial empirical support from the multi-application PoC; the mapping table in Section 1.6.1 reflects this allocation. Chapter 9 examines the AI-assisted development approach that enabled this evaluation at scale, Chapter 10 revisits the research questions with the combined evidence, and Chapter 11 synthesises findings, limitations, and future work.

Chapter 9

AI-Assisted Development: Authorship, Process, and Reflection

This thesis was conceived, designed, evaluated, and written by the author. It was *built* with substantial help from an AI agent during a focused twelve-day productisation sprint near the end of the project. This chapter draws that distinction precisely: it assesses authorship (Section 9.1), describes the knowledge protocol and persistent memory behind the human–agent collaboration (Sections 9.2 and 9.3), and closes with reflection and the relation of the framework to autonomous-agent feedback loops (Sections 9.4–9.7).

The chapter makes three claims: the author's intellectual ownership—over the SDT framework, the XSDLC architecture, the evaluation design, and the thesis narrative—is traceable through the version-control record; the author-designed CLAUDE.md knowledge protocol is what allowed an AI agent to accelerate implementation without dictating direction; and SDT's structured output positions it as a natural feedback source for autonomous-agent loops, to be tested further as future work (Section 9.6.2).

9.1 Authorship and the Author's Role

This section answers who did what in this thesis. Every claim below is reproducible from the project's git history, the persistent agent-memory directory at `~/.claude/projects/-home-meter-peter-development-driveby/`, and the sixteen `CLAUDE.md` files in the monorepo.

9.1.1 The Two Phases of Development

The repository's commit history makes a clean distinction between two phases.

Phase A — Solo development (3 April 2025 – 18 March 2026, 349 days). During this phase the author worked alone. The DDT/SDT axioms were drafted, the nine principles were enumerated, the `APISpec` abstraction was designed, the first principle checkers were written, the OpenAPI 3.x and Swagger 2.0 adapters were built, the validation modes were specified, the monorepo was restructured, the five evaluation APIs were designed and implemented, and the Crossplane composition was prototyped. All twenty-two commits in this phase carry no `Co-Authored-By` trailer; they are entirely the author's work. No generative AI assistance was used in this phase—neither an agentic tool nor a Copilot-style IDE assistant; development used a conventional IDE whose code intelligence was limited to non-generative tooling (symbol completion, navigation, linting).

Phase B — Author-directed AI partnership (19–30 March 2026, 12 days). During this phase the author engaged Claude Code [2] intensively for productisation: the principle checking was finalised and DriveBy was brought to its released form, the GitOps pipeline was wired together, the multi-API evaluation tooling was developed, the XSDLC composition was hardened, and the thesis manuscript was drafted under the author's chapter specifications and feed map (Section 9.5). Twenty-six of the fifty-two commits in the project carry the `Co-Authored-By: Claude` trailer; all fall in this 12-day window. Human-only commits continued throughout this phase—the last (2026-03-29) one day before the final commit—so the author kept making manual code changes throughout the partnership.

This timeline matters. The conceptual content of the thesis—the axioms, the principles, the architecture, the evaluation design—predates the AI partnership by nearly a year. The AI's role was not to invent the framework; it was to accelerate the implementation and writing of a framework that already existed in design.

9.1.2 Quantitative Authorship Evidence

Table 9.1 reports the raw authorship statistics drawn from `git log -numstat`.

Although the commit count is split evenly, the author authored almost three quarters of all source lines added across the project's lifetime: the co-authored commits are a productisation surface on top of an author-built foundation. The roughly 29,000 removed lines trace deliberate design iteration, not churn: the v3.0.0 monorepo restructuring, the deletion of the obsolete `samples/` directory, and the mid-project redesign to the two-repo BYOCI

Table 9.1: Authorship statistics across the entire project (52 commits, 3 April 2025 – 30 March 2026).

Metric	Human-only	Co-authored	% human-only
Commits	26	26	50.0%
Lines added	125,305	44,669	73.7%
Lines removed	21,330	7,678	73.5%
Phase	both A and B	only B	—

architecture (Chapter 7) each removed superseded code in bulk.

Code quality rules authored solo. The five mandatory project-wide invariants (enumerated in Section 9.2.2) are listed in the root `CLAUDE.md` and were authored during Phase A. They constrained every later AI-generated edit; the agent did not negotiate them, the agent obeyed them.

The maintenance rules and the CLAUDE.md hierarchy (Section 9.2) are author-designed. The hierarchy of sixteen `CLAUDE.md` files was authored solo during Phase A; subsequent updates during Phase B were verified by the author before being committed.

The persistent-memory feedback corrections are the author teaching the agent, not autonomous learning: the author authored every one of the twelve entries (Section 9.3); the agent obeyed them.

9.1.3 Effort Allocation

How the author spent Phase B is broken down below, as author estimates from session notes and the persistent-memory log, not exact wall-clock measurements.

Roughly 70% of the author's time during Phase B went to *directing and verifying*: scoping work, reviewing outputs, maintaining the knowledge protocol, and persisting corrections. The remaining 30% went to direct hands-on work, which produced the four human-only commits made during Phase B.

How debugging was carried out. The "manual code changes, debugging, hot-fixes" row combined three methods: the Go test suite (`go test ./test/...`), where the author wrote a failing test before fixing so the fix is regression-protected; manual reproduction against the five evaluation APIs or the live cluster; and artefact inspection—JSON validation reports, Crossplane composition status, controller logs—to localise faults in the GitOps pipeline, where no unit test reaches.

Table 9.2: Approximate author effort allocation during the 12-day Phase B productisation sprint (author estimates; precise wall-clock data was not recorded).

Activity	**% of Phase B**
Architectural direction and task scoping (author)	25%
Reviewing AI-authored code, manifests, and prose (author)	30%
Writing and updating CLAUDE.md and feed-map files (author)	15%
Manual code changes, debugging, hot-fixes (author)	15%
Drafting feedback and persisting corrections (author)	10%
Cluster operations (kubectl, ArgoCD, debugging GitOps state)	5%

Forms of feedback supplied to the agent. Feedback took four forms, in increasing order of permanence: (i) in-session corrections; (ii) commit-message reviews and amendments before pushing; (iii) edits to the relevant `CLAUDE.md` file when a recurring class of mistake was identified; (iv) entries in the persistent-memory directory (Section 9.3) for cross-session behavioural rules. This escalation hierarchy is itself a contribution of the agent-driven development framework: corrections move from one-shot to permanent based on how often the same mistake recurs.

Manual review of almost-final artefacts. Beyond the automated CI checks, the author manually reviewed every committed artefact: every Crossplane composition against live cluster behaviour, every evaluation result against the underlying JSON, every thesis chapter end-to-end, every commit message before publication. The automated checks catch structural drift; the manual review catches content errors that no test can.

Report-side effort. Of the 30% in Table 9.2 spent reviewing AI-authored outputs, roughly half—about 15% of total Phase B effort—went to the thesis manuscript itself: checking each chapter against the source code and evaluation data, correcting factual and numerical claims, and rewriting substantive passages (Section 9.5). Every chapter received at least two full author read-throughs before commit; this too is a session-notes estimate.

9.1.4 Overall Authorship Claim and Role

Combining the conceptual ownership (the framework, the architecture, the evaluation design, the thesis report structure), the quantitative line-level evi-

dence (approximately 74%—73.7% of lines added and 73.5% of lines removed—authored solo, per the table above), the timeline (a year of solo work followed by 12 days of directed partnership), and the reviewer-of-record role for every artefact, the author claims overall authorship of approximately **80%** of this thesis and the DriveBy project. The remaining 20% is acknowledged AI co-authorship, concentrated in the Phase B productisation surface and in initial drafts of several thesis sections. The author—not the agent—rewrote, corrected, and finalised the content of every chapter, so the final text of the thesis is in every case author-authored (Section 9.5).

Role in software-project terms. The author's role maps most cleanly to *architect-and-tech-lead*: defining the technical vision (the SDT axioms and principles), designing the system architecture (the `APISpec` abstraction, the principle-checker pattern, the XSDLC composition, the two-repo BYOCI model), establishing the five code-quality invariants, and reviewing every output before integration. During Phase B the role expanded to explicit *agent orchestrator*: deciding what to delegate, supplying the structured context the agent required, and accepting or rejecting each output. The author did not act primarily as an analyst or as an implementor.

Did the original idea survive? The original idea—a framework that derives validation rules from the API specification rather than from manually authored tests, with a CLI that integrates into a GitOps pipeline—was implemented in full. Two notable scope changes occurred during the project, both author-driven decisions made before Phase B:

- The framing was sharpened to a *conceptual framework* rather than a complete formal methodology (Chapter 3; also reflected in Chapter 1). This is a sharpening of the original claim, not a deviation from it.
- The Kubernetes integration was redesigned mid-project from a single-repo model into the two-repo BYOCI architecture (Chapter 7). This change was driven by an architectural insight about source hydration that the author identified independently; it expanded the contribution rather than replacing it.

No deviation in the project was driven by the AI agent suggesting a more interesting pathway. The Phase B partnership accelerated execution of the design that already existed; it did not redirect the design.

9.1.5 Demarcation of Intellectual Contribution

The quantitative evidence above answers "how much" was authored solo; this subsection answers "what specifically," because in an academic defence

the burden of separating one's own contribution from agent-assisted artefact production rests with the candidate.

Author's intellectual contribution. The conceptual and design content of this thesis, traceable to author-only commits and to design decisions made before or independently of any AI session:

- The **SDT paradigm** as a whole—the three framework-wide axioms (Completeness, Determinism, Observability), the nine validation principles (P001–P009), the validation modes (`minimal`, `strict`, `test-ready`, `test-only`), and the explicit rejection of an axiom-to-principle partition in favour of axioms as framework-wide invariants (Chapter 3).
- The **formal model of validation as parallel application and aggregation**, $f_p(s) = \text{agg}(f_{p_1}(s), \ldots, f_{p_n}(s))$, with the explicit non-composition claim and its three consequences for Determinism, modular modes, and concurrent evaluation (Section 3.3.1).
- The **DriveBy architecture**: the `APISpec` abstraction layer, the atomic-principle design and `PrincipleChecker` interface, the deterministic engine and registry pattern, the structured-report contract (Chapter 5).
- The **XSDLC composition design**: a single declarative CRD reifying the full delivery pipeline, the two-repo BYOCI boundary, the per-environment overlay model, the promotion-chain mechanics (Chapters 6 and 7).
- The **two-tier closed-loop quality-control framing** of the gate: verdict tier (open-loop) versus platform tier (closed-loop, autonomous repair across the controller stack), and the claim that drift between gate evaluations is blocked by construction rather than detected after the fact (Section 7.6.2).
- The **evaluation methodology**: the four-arm design (controlled, large-scale, operational, agent-feedback), the principle-injection scheme against the five evaluation APIs, and the single-iteration agent-feedback experiment (Chapter 8).
- The **five project-wide code-quality invariants** (Section 9.1.2) and the `CLAUDE.md` hierarchy that operationalises them as constraints on every later agent-assisted edit (Section 9.2).
- The **thesis chapter structure** and the chapter-to-source-code feed map (`thesis/CLAUDE.md`), which together define what each chapter argues and which source artefacts it must remain faithful to.

Agent-assisted scope. The agent contribution, concentrated in the 12-day Phase B and bounded above by the author's review and acceptance of every committed output:

- **Routine Go boilerplate** within the principle checkers, authored against the interface and types the author had already designed. The conceptual content of each checker (what to check, why, with what severity) was author-specified; the agent rendered it as Go.
- **LaTeX typesetting and prose-level revisions**: first drafts of prose around author-supplied chapter outlines and feed-maps; the author reviewed, edited, and committed.
- **Documentation scaffolding**: per-package `CLAUDE.md` stubs and READMEs, and the documentation-completeness CI check. The hierarchical protocol itself and the root `CLAUDE.md` are author-designed.
- **Operational tooling**: batch-validation scripts for the APIs.guru harvest, log-parsing helpers, the evaluation-table harness, and the per-PR comment formatter. The evaluation methodology these scripts serve is author-designed.

The boundary in one line. Every architectural, formal, and methodological decision in this thesis was authored, reviewed, and committed by the candidate. Agent assistance is a development methodology—a way of accelerating the realisation of a design that already exists—not a substitute for the conceptual contribution that defines what is being realised.

9.2 The CLAUDE.md Knowledge Protocol

The central enabling infrastructure for the Phase B partnership—and the author's most reusable methodological contribution from the agentic-development framework (Contribution 6 in Chapter 1)—is a hierarchical system of `CLAUDE.md` files distributed across the monorepo. These are neither READMEs nor configuration but agent context files: structured knowledge artefacts consumed by AI agents at the start of every session, providing the persistent context that enables cross-session continuity.

9.2.1 Hierarchical Structure

The DriveBy project contains sixteen CLAUDE.md files organised in a tree that mirrors the monorepo structure (Table 9.3):

Table 9.3: CLAUDE.md hierarchy in the DriveBy monorepo.

File	Scope
`CLAUDE.md` (root)	Project vision, SDT axioms, monorepo map, build commands, code quality rules, maintenance rules
`driveby-cli/CLAUDE.md`	Package map, dependency flow, validation modes, test structure
`driveby-cli/internal/principles/CLAUDE.md`	PrincipleChecker interface, per-file status, axiom assignments, procedure for adding principles
`driveby-cli/internal/spec/CLAUDE.md`	APISpec interface, adapter pattern, normalised types, strict rules
`kubernetes/CLAUDE.md`	Helm chart structure, XSDLC schema, cluster state, template escaping
`apis/CLAUDE.md`	API inventory, evaluation purpose
`apis/perfect-api/CLAUDE.md`	Endpoints, security model, expected pass/fail results
`apis/bad-docs-api/CLAUDE.md`	Endpoints, deliberate documentation/security gaps, expected failures
`apis/broken-api/CLAUDE.md`	Endpoints, implementation divergences from spec, expected P006 failures
`apis/no-auth-api/CLAUDE.md`	Endpoints, deliberate security omissions, expected P005 failures
`apis/slow-api/CLAUDE.md`	Endpoints, artificial latency injection, expected P007 failures
`thesis/CLAUDE.md`	Chapter specifications, source-to-code feed map, figure checklist, LaTeX conventions
`docs/CLAUDE.md`	Documentation index, diagram coverage
`.github/CLAUDE.md`	CI/CD workflow descriptions
`tools/CLAUDE.md`	Batch testing workflow, CSV formats
`results/CLAUDE.md`	Result directory structure, gate outcome format, verification matrix

When an agent begins working in a directory, the CLAUDE.md files in the path from root to that directory are loaded into its context—e.g. an agent modifying `driveby-cli/internal/principles/p004_schema.go` receives the root, CLI, and principles files. This layered loading gives the agent both the global invariants and the local domain knowledge for the task at hand.

Hierarchy depth and rationale. The hierarchy is at most three levels deep (root → package → component), a trade-off chosen empirically during Phase A: a flat structure left sessions without package-level context, while a five-level structure produced redundant content the agent's context window could not absorb without truncation.

Containment vs. repetition. Some sub-directory CLAUDE.md files repeat invariants from the root file. An alternative design would state each invariant exactly once in the highest applicable file and let layered loading propagate it downward; this design was considered and rejected in favour of repeating the five *critical* code-quality rules in the leaf files. The justification is observational: in the Phase A depth experiments, sessions demonstrably truncated ancestor content, whereas the file being actively edited was never truncated—the leaf file is the only placement observed to survive context-window pressure. The cost is duplication; the benefit is empirically needed robustness.

9.2.2 Content Categories

Each CLAUDE.md file contains four categories of knowledge:

Structural knowledge: what exists, where it lives, and how it connects—the root monorepo directory table, the CLI package dependency graph, the per-principle status table.

Invariants: rules that must never be violated. The root file specifies the five code quality rules: (1) the mandatory `spec.APISpec` abstraction; (2) the `PrincipleChecker` interface for every checker; (3) no `os.Exit()` in command handlers; (4) types confined to `internal/types/` with no circular dependencies; (5) CLI configuration shared via `config.go` with no duplication. These are repeated in leaf files per the placement rationale of Section 9.2.1.

Procedures: step-by-step instructions for common tasks—adding a new principle, building, testing, releasing—transforming tribal knowledge into structured, machine-executable instructions.

Thesis mapping: bidirectional traceability between code and academic writing. Every sub-directory CLAUDE.md names the thesis chapter describing that code; the thesis CLAUDE.md's Complete Feed Map maps every thesis section to its code and documentation sources. This enables both updating

thesis content when code changes and updating code when thesis analysis reveals inconsistencies.

9.2.3 Maintenance Rules and Self-Enforcement

The CLAUDE.md protocol includes four maintenance rules encoded in the root file:

1. When modifying source files, update the corresponding CLAUDE.md in that directory.
2. When modifying source files, update the corresponding `docs/` file if documented behaviour changes.
3. When adding a new principle, create: source file, test file, `docs/principles/PXXX.md`, and update `principles/CLAUDE.md`.
4. When adding a new directory, create a CLAUDE.md in it and add it to the root monorepo table.

One clarification on the fourth rule: directories were planned up-front by the author during the monorepo restructuring, so the rule was exercised mainly as a verification step; it would matter more in projects where AI agents drive the directory layout.

These rules are not merely advisory: the CI pipeline includes a documentation completeness check (`tools/check-docs.sh`) performing nine categories of checks—CLAUDE.md existence, principle documentation completeness, documentation-versus-reality consistency (detecting stale claims like "Not yet implemented" when the source file exists), thesis chapter status, and test coverage gaps—and fails the build on any violation. This closes the loop: the protocol tells agents what to update, the CI check verifies updates were made, preventing knowledge drift—the same problem SDT prevents for API specifications, by the same means: treat documentation as a testable artefact and gate on its completeness.

9.3 Persistent Agent Memory

Beyond the per-directory CLAUDE.md files, the project maintains a persistent memory system that survives across agent sessions. This memory captures information that cannot be derived from the codebase: user preferences, learned corrections, project decisions, and external references.

Category of memory system. In the taxonomy of memory architectures for AI systems, this is *external, file-backed, semantic* memory: state lives outside the agent process in version-controllable Markdown files storing generalised knowledge—rules, preferences, project facts—rather than conversation transcripts. It is neither episodic (no per-session logs) nor procedural (procedures live in the CLAUDE.md files).

The memory directory contains twelve files: an index file (`MEMORY.md`) and eleven content entries.

- **User profile** (1 file): the developer's role (diploma thesis student), expertise (deep Go and Kubernetes, new to LaTeX), and communication preferences.
- **Feedback corrections** (8 files): learned behaviours from iterative interaction, e.g. `feedback_docs_updates` ("always update CLAUDE.md, docs, and thesis drafts after each implementation session") and `feedback_runtime_spec` ("sample APIs generate specs at runtime, not from static `openapi.json`"—a correction made after the agent assumed file-backed specs).
- **Project decisions** (1 file): the monorepo restructuring rationale.
- **External references** (1 file): the target cluster URL and the instruction to never reinstall ArgoCD.

The feedback corrections are the most active part of the system. Each encodes a lesson from a failed interaction, persisted so the same mistake would not recur; over Phase B these accumulated into a behavioural profile that shaped the agent's approach. This is analogous to organisational learning in human teams [48]: tacit knowledge is externalised into explicit, persistent artefacts that new sessions inherit.

A clarification on what this memory is *not*: it is not an autonomous learning system. The agent does not write to memory on its own initiative; the author writes every entry, the agent obeys. The persistence mechanism is what compounds the corrections, not any agentic learning capability.

9.4 Agent-Driven Multi-API Evaluation

The evaluation in Chapter 8 was not conducted manually. All four arms—controlled defect-injection, large-scale APIs.guru, operational deployment, and the agent-feedback experiment—were designed by the author. The APIs.guru arm, the focus of this section, was tooled and executed through an agentic workflow during Phase B and analysed jointly.

9.4.1 Tooling Development

The three-step batch testing pipeline (`tools/harvest-openapi-apisguru.py`, `tools/probe-openapi-endpoints.py`, `tools/run-openapi-batch.sh`) was developed collaboratively: the author specified the experimental protocol (harvest from APIs.guru, filter for reachable endpoints, validate in both minimal and strict modes) and reviewed every script before commit; the agent implemented the tooling under that specification. Reachability filtering is necessary even though the static specification is the object of validation: it excludes abandoned registry entries so the reported pass rates reflect currently operated services, and it is a precondition for P006–P007, which require a responding host.

This tooling enabled evaluation at a scale impractical manually, producing the per-principle pass rates reported in Section 8.5 of Chapter 8.

9.4.2 Agent as Experimenter

The agentic workflow extended beyond tooling development. The agent executed the evaluation runs, drafted summary tables of the results, and identified candidate patterns (e.g. the universal gap between P001 pass rates and P002–P004 pass rates). The author reviewed every analysis, verified the numbers against raw output, refined the interpretation, and is responsible for every numerical claim in Chapter 8. The agent did not interpret the results in the thesis; the author did.

This workflow illustrates a key property of SDT's structured output: the same structured, machine-readable, deterministic reports that suit CI/CD quality gates also suit agent-driven evaluation under human supervision. The Axiom of Observability (Chapter 3) is what enables this workflow.

9.4.3 The Limit Acknowledged: Coverage of Per-Check Cases

Exercising only one or two cases per principle would leave most of the check matrix untested. The controlled evaluation in Chapter 8 therefore extends beyond whole-principle defects to per-check injection—two to three targeted mutations per implemented principle, fifteen in total, plus selected combinations of two and three simultaneous defects. This samples the check space rather than exhausting it (Chapter 8 records complete per-check coverage as future evaluation work), but it demonstrates that detection accuracy holds across every principle at the granularity of individual checks. The injection

scheme—which defect at which check, with what expected verdict—was specified by the author; the agent implemented the injection harness under that specification, and the author reviewed every injected defect and verified every verdict before accepting the runs.

9.5 Agent-Driven Thesis Writing

The crucial fact first: the agent's contribution to the thesis was confined to initial, code-grounded drafts; after each draft the author took over completely, fully rewriting substantive sections, so the final content of every chapter is the author's. Within that model, the Phase B agentic workflow divided labour as follows:

- **Author**: chapter structure (the eleven-chapter outline of `thesis/main.tex`), every research question, every contribution, every figure design, every claim that maps to a specific code element, every numerical result, and the final edited form of every chapter.
- **Agent**: first drafts of prose paragraphs against author-supplied chapter specifications and the feed map; LaTeX formatting; cross-reference plumbing; mechanical edits (e.g. the SDT renaming pass).
- **Both**: identification of inconsistencies between thesis claims and code reality. The agent could query the codebase faster, but the author judged whether each flagged inconsistency was a thesis error, a code error, or an acceptable simplification.

The thesis CLAUDE.md file (202 lines) provides chapter-by-chapter specifications that function as agent task descriptions: expected sections, source files to read, and key content requirements, with the Complete Feed Map mapping every thesis section to its code and documentation sources. When drafting a section the agent follows a structured protocol:

1. **Read the specification**: the chapter entry in the thesis CLAUDE.md.
2. **Read the source**: the code and documentation files named in the feed map.
3. **Draft the content**: LaTeX grounded in the actual implementation as the authoritative source.
4. **Author takeover**: the author rewrites substantive sections, edits for accuracy and tone, and accepts or rejects.

Thesis as specification: a worked example. The thesis statement that "DriveBy implements seven of nine principles as `PrincipleChecker` instances, with P006 and P007 as planned wrappers" is treated as authoritative: the CI documentation check cross-references it against the actual files in `driveby-cli/internal/principles/`, and if a P006 file appeared in source without a corresponding thesis-status update, the build would fail. The thesis is thus the source of truth for the project's claimed scope, and the code is checked against it—more limited than treating the entire thesis as formal specification, but formalising one claim per chapter as a CI-enforced invariant.

9.6 SDT as Agent Feedback Infrastructure

The experience of building DriveBy with an AI agent supports a claim made throughout this thesis: that SDT's structured output is naturally suited for consumption by autonomous agents. This section formalises the connection and notes its empirical status.

9.6.1 The Agent Validation Loop

Consider an autonomous agent tasked with improving an API specification's SDT compliance. The agent executes the following loop:

1. **Validate**: run `driveby validate-only -validation-mode strict`.
2. **Parse**: read the JSON report; identify failing principles and check failures.
3. **Diagnose**: read each failure's `SuggestedFix` field and `Details` map.
4. **Remediate**: generate specification changes addressing the diagnosed issues—missing descriptions (P002), error responses (P003), schema constraints (P004), security schemes (P005).
5. **Re-validate**: run DriveBy again to confirm the fix without new failures.

This loop is feasible because of three properties guaranteed by SDT's axioms:

- **Completeness** ensures the report identifies *what* is missing: a P002 failure listing "44 request/response bodies missing examples" gives the agent a precise remediation target.

- **Determinism** lets the agent distinguish genuine fixes from noise: on re-run, a P002 pass confirms the fix; a continued failure indicates it was incomplete.
- **Observability** ensures the report is structured and machine-readable—the agent reads fields, not prose.

9.6.2 Empirical Evidence and Open Validation

The theoretical connection above between SDT feedback and agent corrections requires empirical demonstration; the development experience alone is indirect evidence—it shows SDT's output is consumable by an agent under human supervision, but not that an agent can autonomously improve a specification's SDT score using only the validation report as input.

Such an experiment is reported in Chapter 8 (Section 8.6): the Swagger Petstore specification—a real-world public API scoring 1/6 in strict mode—is taken as the starting point, the DriveBy validation report is supplied as the sole input to an AI agent, and the agent's revised specification is re-validated. A single agent-revision pass raised the score from 1/6 to 4/6, converting the theoretical connection of this section into an empirical one. The author considers a more comprehensive version of this experiment (multiple agents, multiple specification revisions per API, controlled comparison against human-revised specifications) to be an important direction for follow-up doctoral work (Chapter 11).

9.6.3 From Validation to Remediation

The `SuggestedFix` field in each `PrincipleResult` is the interface between validation and remediation: it names the specific operations, schemas, fields, or security definitions at fault. These fixes are not aspirational—in the agent-feedback experiment (Section 8.6) the agent translated the `SuggestedFix` entries of the baseline Petstore report into concrete edits: descriptions and examples (P002), 4xx/5xx error responses wired to all 19 operations (P003), schema constraints (P004), security schemes (P005), and versioning metadata (P008). When DriveBy reports "string field `email` lacks length constraints," the remediation is unambiguous: add `maxLength` and `pattern` to the field's schema, derivable from the report without external documentation, a human, or inferred intent.

That the validation report contains sufficient information for autonomous remediation is the Axiom of Completeness applied to the validation output rather than the specification input; the `SuggestedFix` field ensures it.

A limitation: invented constraint values. One caveat tempers the remediation claim. An agent that supplies concrete constraint values—a `maxLength` bound, a `pattern` regular expression—without grounding them in the actual implementation can produce a specification that *passes* the principles while *misrepresenting* the API's real behaviour. The divergence is not necessarily detectable even by functional testing (the implementation may accept inputs the invented constraint forbids), and such a specification would mislead API clients. Agent-remediated specifications must therefore be verified against the implementation by its owners before promotion to production; automating this verification (e.g. cross-checking proposed constraints against observed traffic or implementation code) is left as future work.

9.7 Personal Lessons Learned

Whereas lessons about the agentic-development *framework* generalise (Sections 9.2 and 9.3; Chapter 10), this section records what *the author* learned from this specific experience.

The 350-day solo investment was decisive, not the 12-day partnership. The AI partnership only became productive because of the year of solo work that preceded it: without the SDT axioms, the principle catalogue, the `APISpec` abstraction, the code-quality rules, and the monorepo layout already in place, Phase B would have produced architecturally inconsistent code. In a follow-up project the author would again resist engaging AI assistance early. Could the agent also have supplied design ideas worth adopting? In this project it did not: the division of labour was deliberately strict—architect and executor—all architectural and design decisions originated with the author, and the agent's suggestions stayed at implementation-level tactics, none adopted as a design-level idea (Section 9.1.4).

Persisting corrections is the highest-leverage skill. The behaviour that made the largest difference was treating every recurring agent mistake as a CLAUDE.md or memory entry rather than a one-shot correction: a correction that lives in a chat session dies with the session; one that lives in a file compounds.

Code review remained the bottleneck. The agent could produce more output than the author could verify; the verification rate was the rate-limit. In a follow-up project the author would budget explicit verification time—reviewing the previous session's output to completion before starting the next.

The thesis benefited from being treated as one of the artefacts. Drafting the thesis with the same protocol used for code (chapter specifications,

feed map, CI documentation check) produced tighter alignment between document and implementation than a separate writing process would have; the author would adopt this pattern by default in follow-up projects.

What the author would do differently. Three specific changes for a follow-up project are planned: (i) record exact wall-clock time per activity from day one (so that questions like "how much time did you spend reviewing AI output?" have measured rather than estimated answers); (ii) set up the controlled-experiment harness for the SDT-feedback evaluation (Section 9.6.2) before starting Phase B, so that the empirical proof of the framework's most consequential claim is part of the deliverable from the start; (iii) establish the precision/accuracy axiom debate (Chapter 3) earlier, so that the framework is presented as four-axioms-with-precision rather than three-axioms-with-precision-as-future-work.

Not every lesson is AI-specific. Three transferable practices carry forward irrespective of agentic tooling: fixing architectural invariants up front as non-negotiable constraints; treating documentation as a CI-gated, testable artefact (the practice SDT itself generalises); and maintaining authorship traceability through disciplined commit metadata, which made the quantitative analysis of Section 9.1.2 possible at all.

9.8 Summary

The author owns the conceptual content of this thesis and approximately 74% of all source lines added across the project's lifetime; the AI partnership was concentrated in a 12-day productisation sprint at the end of a 361-day project, during which the author acted as architect, agent orchestrator, and reviewer of every output. The CLAUDE.md knowledge protocol and the persistent-memory system made that directing role tractable, and SDT's structured output as agent feedback is demonstrated empirically in Chapter 8. The author claims overall authorship of approximately 80% of this thesis and the DriveBy project, in the role of architect-and-tech-lead with explicit agent-orchestrator responsibilities during the productisation sprint.

Chapter 10

Discussion

This chapter revisits the four research questions with evidence from the evaluation (Chapter 8) and the agentic development experience (Chapter 9), reflects on the broader implications of the results, and identifies the tensions and trade-offs that emerged during design and deployment.

10.1 Revisiting the Research Questions

10.1.1 RQ1: Framework Soundness

Can a conceptual framework grounded in specification completeness, behavioural determinism, and operational observability demonstrate the feasibility of automated API quality assurance that can operate without dedicated testers in the promotion pipeline?

The principle-level arm of the controlled evaluation demonstrated 100% detection accuracy for its seven injected defects with zero false positives across all seven implemented principle checkers. At the finer check level the figure is lower: the per-check extension (Section 8.3.4) detected twelve of fifteen single-check defects (80%); of the three non-detections, one mutation was neutralised by the loader's structural validation, one proved an injection artefact with no real defect, and exactly one is a genuine framework miss. Combined defects spanning multiple principles—three random pairs and one triple—flagged every targeted principle concurrently in all evaluable cases, confirming checker independence. Detection was deterministic: repeated runs against the same defective specification produced identical results.

However, "operating without dedicated testers" requires qualification. SDT detects a specific class of defects: documentation completeness gaps, schema imprecision, security misconfiguration, and versioning discipline failures. It

does not detect business logic errors, performance regressions under realistic load patterns, or usability issues that a human tester would identify through exploratory testing. SDT replaces the *systematic, repeatable* aspects of manual API review—the checklist that a QA engineer follows—while leaving the *creative, exploratory* aspects to human judgment.

Each of the three axioms is backed by evaluation evidence rather than design rationale alone. Completeness: real-world specifications (Section 8.5) systematically lack the descriptions, examples, error responses, and security schemes that SDT requires; without P001–P004 these gaps would go undetected. Determinism: the quality gate operates in an automated loop where non-deterministic results would oscillate the gate between pass and fail; the controlled evaluation verified the property empirically (Section 8.3). Observability: the CI/CD pipeline consumes exit codes, the AI agent parses JSON reports, and the agent-feedback experiment (Section 8.6) demonstrated that the JSON report alone is sufficient feedback for an autonomous agent to repair a failing specification.

The five-API controlled evaluation (Section 8.4) extended this finding to a multi-API setting: specification-level defects (`bad-docs-api` on the critical P002–P004, `no-auth-api` on the critical P005) were blocked at the staging gate, while behaviourally defective APIs with clean specifications (`broken-api`, `slow-api`) passed static validation and were caught at the runtime layer by P006 and P007 respectively. This layered discrimination demonstrates that the nine principles are not redundant but mutually complementary: each covers a failure class the others cannot detect.

10.1.2 RQ2: Automation Extent

To what extent can validation rules, test cases, and quality gates be derived automatically from the OpenAPI specification together with declared operational policy (the XSDLC resource), and does the resulting automation produce actionable, machine-readable signals suitable for both human engineers and autonomous agents?

Fully, for validation rules and test cases under the implemented principles. All seven principle checkers derive their validation logic entirely from an OpenAPI specification, and the large-scale evaluation confirmed meaningful results for any valid specification without per-API configuration, custom rule files, or manual test case authoring. This zero-test-authorship property is the central practical contribution of SDT: it eliminates the maintenance burden of separate test suites for API quality assessment. Quality gates, however, are *not* derived solely from the OpenAPI specification: the environment chain, the checks composing each gate, and the load-testing thresholds are supplied

by a second declarative artefact, the XSDLC custom resource (Chapter 6).

The limitation is scope. SDT validates specification *quality*—completeness, precision, consistency—but does not verify, as hand-authored integration tests do, that the API's business logic produces correct results for domain-specific inputs. The functional testing subsystem (P006, implemented but not yet wrapped as a PrincipleChecker) bridges part of this gap by verifying that responses match declared schemas. The boundary deserves precision: the specification is not entirely silent on business logic—operation descriptions, parameter constraints, enumerations, and request/response examples encode fragments of it, and P006 can in principle assert any constraint the schema language can express (value ranges, formats, required fields, example-conformant responses). Out of reach is logic that no OpenAPI construct can state—stateful workflows, cross-call invariants, and domain rules relating inputs to computed outputs—which would require an out-of-band oracle. It is this residue, not business logic wholesale, that still demands manual test authorship.

The agent-driven evaluation (Chapter 9) addressed the second half of the question: whether the spec-derived signals are actionable for autonomous consumers. The AI agent consumed DriveBy's JSON reports directly, identified cross-API patterns in the APIs.guru dataset, and generated comparative analysis; the structured output—per-principle pass/fail, specific check failures, suggested fixes—was sufficient without human interpretation. This establishes the *sufficiency* of the structured output for autonomous consumption; the completeness and accuracy of the checkers themselves rest on the controlled evaluation. Together they validate the Observability axiom's practical utility beyond CI/CD gating.

10.1.3 RQ3: GitOps Integration

How effectively does specification-driven quality assurance integrate into cloud-native GitOps pipelines, and can a single declarative custom resource replace the manual provisioning of quality gate infrastructure?

The proof-of-concept deployment on `private.novelcore.org` demonstrated effective integration. A single XSDLC custom resource generates approximately 45–50 managed Kubernetes objects in under 15 seconds, replacing a manual ClickOps process that takes hours, is error-prone, and cannot be version-controlled. The CI/CD integration metrics (Section 8.2.8) quantify the contrast: sub-second CLI startup, 2–5 second validation runs, $\sim$15 s from CR apply to all resources ready, and under 90 seconds end-to-end from webhook receipt to posted commit status—latencies compatible with interactive developer feedback loops. The system was installed into an active

cluster alongside the existing Argo stack without modifying any pre-existing infrastructure, and can be removed without affecting it.

The proof-of-concept quality gate demonstrated the feasibility of operating without dedicated testers in the promotion pipeline. Every promotion PR was automatically validated against the specification contract and, where configured, load-tested against performance thresholds. The gate fired without human initiation, ran without human test authorship, and produced results without human interpretation. While this validates the architectural approach, production adoption would require further evaluation at organisational scale.

Two tensions emerged during deployment. First, the *event pipeline reliability* tension: webhook delivery from GitHub to the in-cluster EventSource traverses multiple network boundaries, and any failure in this chain silently drops the event. The EventBus provides at-least-once delivery within the cluster, but external webhook delivery is at-most-once; GitHub's webhook retries mitigated this in practice, but a production-grade deployment should consider a webhook receiver with explicit acknowledgment.

Second, the *composition complexity* tension: the Crossplane composition that generates WorkflowTemplates operates across four nested template layers (Chapter 6), each with its own escaping rules, so debugging the rendered output requires understanding all four simultaneously. This complexity is inherent to the meta-controller pattern—generating complex infrastructure from minimal input comes at the cost of debugging opacity.

The multi-API deployment reinforced this finding: five `XSDLC` custom resources ran simultaneously on the same cluster without interference, each API's gate operating independently under the per-namespace isolation model (Chapter 8)—validating XSDLC's scalability from a single-CR demonstration to a multi-application platform feature.

10.1.4 RQ4: Ontological Specification

Can quality-gated delivery pipelines be expressed as Kubernetes-native ontological specifications that operators and autonomous agents can discover, inspect, and manage as composable building blocks within an internal developer platform?

Yes. The XSDLC CRD functions as a machine-readable, structurally typed specification that operators and controllers can discover via the Kubernetes API, inspect for configuration, and manage as a composable building block. In Novelcore's IDP, it sits alongside database provisioning compositions, networking configurations, and observability stacks—all following the same Crossplane pattern.

The ontological framing proved practically valuable in two ways. First,

it enabled the `kubectl get xsdlcs` summary view with printer columns, making the quality gate system observable at the platform level. Second, it positioned XSDLC as a child resource candidate for higher-level operators: a microservice provisioning controller could create an XSDLC resource when onboarding a new service.

The limitation is that the ontological properties are available but not yet exercised by autonomous consumers. No production controller currently creates or modifies XSDLC resources programmatically. Such management would not be a new control loop: it is the same declare-observe-converge GitOps reconciliation applied to a second specification layer—the pipeline contract itself rather than the API's content contract (the OpenAPI document). The CRD's structural typing and API discoverability make this feasible, but the integration point remains theoretical. Future work (Section 11.5 of Chapter 11) identifies adaptive load testing thresholds and agent-driven remediation as concrete use cases for programmatic XSDLC management.

10.2 The Specification Quality Gap

10.2.1 The OpenAPI Specification Quality Gap

The large-scale evaluation revealed a consistent pattern: documentation quality and constraint precision are systematically low, and even structural compliance is weaker than expected. APIs in the sampled APIs.guru population fail P002, P003, and P008 universally, with P004 and P005 failing near-universally; structural compliance (P001) holds for fewer than half of the OpenAPI 3.x sample (23/50), although it is universal in the converted Swagger 2.0 arm. Even the Swagger Petstore v3—the canonical OpenAPI example maintained by the OpenAPI Initiative—scores 1/6 in strict mode.

This gap has two implications. First, it validates SDT's thesis that existing tooling validates specification *form* but not *substance*. Linters like Spectral enforce structural rules (valid syntax, no duplicate operationIds) but do not assess whether the specification provides the information a consumer needs (descriptions, examples, error responses, security definitions). Structural linting rules are not even sufficient for full *structural* validation—rule sets cover only the properties their authors anticipated—so the form/substance gap is wider than documentation quality alone. SDT's completeness principles (P002–P004, all critical severity) target precisely this gap—they block promotion when the specification lacks the documentation, error schemas, and type definitions that functional tests require.

Second, it suggests that the gap is *systemic* rather than incidental. The

pattern holds across APIs from major providers and small independents alike. Most likely, existing tooling does not enforce documentation quality, so it degrades under the same pressure that causes documentation drift: when deadlines compress, descriptions and examples are the first casualties because nothing gates on their presence.

SDT changes this dynamic. By gating promotion on specification quality—not just structural validity—SDT creates an economic incentive for completeness. Over time, this shifts the equilibrium from "documentation is optional" to "documentation is a prerequisite for delivery."

10.2.2 XSDLC as Specification of the Specification

The OpenAPI specification is the content contract that SDT validates. The XSDLC custom resource is a parallel specification: a declarative, machine-readable description of the quality-gated delivery pipeline that the Kubernetes API server can itself discover, validate, and reconcile. Where the OpenAPI document specifies the API's *functional* behaviour, the XSDLC document specifies gating policy, including the performance thresholds that the OpenAPI contract cannot express. Together they form a dual-specification model in which both layers are first-class objects of automation, and where the framework's claim of specification-driven quality assurance applies to the pipeline as much as to the service it gates.

This split is contingent rather than fundamental. Performance thresholds are non-functional requirements that arguably belong *in* the API contract itself: an OpenAPI vendor extension such as `x-performance` could carry latency and success-rate targets alongside the operations they govern. Adopting it would make the approach strictly single-artefact—every gated property derived from one document—leaving the XSDLC resource to specify only pipeline topology. This direction is identified as future work in Chapter 11, Section 11.5.

This duality is what distinguishes SDT's gating model from imperative quality scripting. A traditional gate is a procedure encoded in a CI pipeline configuration; it cannot be inspected by anything other than the CI runner that executes it. An XSDLC resource, in contrast, is a structurally typed Kubernetes object: any operator, controller, or autonomous agent can read it, reason about its declared configuration, and update individual fields without re-implementing the underlying composition. The composition mechanics are developed in Section 6.3 (Subsection 6.3.1) of Chapter 6; the broader ontological framing is taken up in Section 11.3 of Chapter 11.

Framing XSDLC as a specification clarifies a distinct contribution: while the empirical work establishes that SDT detects and gates on specification-

quality gaps in OpenAPI documents, the XSDLC contribution sits one layer above—the gate itself, the pipeline that runs it, and the environment topology it traverses are declared in the same specification-first idiom and inherit the same observability, discoverability, and machine-manageability properties. Delivered today: discoverability through the Kubernetes API, inspectability of the declared environment chain and check configuration, composability as an IDP building block, and a Git-native audit trail. Deferred to future work: autonomous controllers that create or tune XSDLC resources programmatically—turning the specification from inspectable into actively managed.

10.3 The BYOCI Boundary

The BYOCI (Bring Your Own CI) principle draws a clean boundary: XSDLC owns delivery (promotion, quality gates, environment management), and the developer owns CI (build, test, image); the boundary and its integration contract are defined in Section 7.3.2 of Chapter 7. It was a deliberate design choice that emerged from observing the failure mode of opinionated CD platforms that attempt to own the full lifecycle.

The boundary proved valuable for adoption. Installing XSDLC required zero changes to the developer's existing CI configuration, repository structure, or branching strategy: any CI system that can update manifests on `main`—GitHub Actions, GitLab CI, Jenkins, a shell script—integrates with XSDLC without modification.

The trade-off is that XSDLC has no visibility into the developer's CI process. It cannot verify that unit tests passed, that the container image was built from the correct commit, or that security scans completed. XSDLC trusts whatever dry manifests appear on the `main` branch. This is by design—the alternative would require XSDLC to understand every possible CI system, which contradicts the composability principle. Consequently, XSDLC's quality assurance is scoped to the specification and performance contracts, not to the full software quality lifecycle.

10.4 Threats to Validity

Internal validity. The controlled evaluation uses a single reference API (`non-critical-api`) into which fifteen check-level defects were injected, plus three random multi-principle pairs and one triple (Section 8.3.4). Multi-principle defects were tested and all targeted principles were detected concur-

rently in 3/3 cases. Three single-check defects were not directly detected; as Section 8.3.4 records, one (`p004-strip-constraints`) was neutralised by the loader's structural validation, one (`p005-strip-op-security`) is an injection artefact with no real defect to report, and only `p009-strip-param-examples` reflects a genuine framework limitation, recorded there and as a candidate strengthening in Chapter 11. The defect set was hand-designed, introducing selection bias toward the implemented checkers: the design cannot reveal classes the checkers miss entirely and may understate the long tail of naturally occurring defects that do not map cleanly to a single check. Mining real defects from public OpenAPI specifications would remove this bias; that mitigation is left to future work. Finally, the multi-API evaluation was conducted with AI agent assistance (Chapter 9); although the agent's analysis was reviewed by the human developer, its tendency to identify thesis-confirming patterns makes the interpretation layer a threat akin to experimenter bias. The raw validation outputs themselves are deterministic and verifiable.

External validity. The large-scale evaluation used 50 public APIs from APIs.guru (Section 8.5.3), spanning OpenAPI 3.0.x and 3.1.0, complemented by a 20-API Swagger 2.0 arm reported in Section 8.5.4. The combined 70-API population reproduces the same systemic documentation-gap pattern across both spec versions: P002/P003/P008 universal fail and P004/P005 near-universal fail—with P001 passing universally in the 2.0 arm but only 23/50 in the 3.x arm, and 84% of 3.x specs and 80% of 2.0 specs scoring 1/6 or worse. The sample is alphabetically biased toward providers whose names sort early (Adyen, Amazon Web Services, Amadeus) and over-represents AWS-style auto-generated specs that typically lack documentation enrichment. This bias cuts both ways: it strengthens the qualitative finding that documentation completeness is the dominant gap, but may also *inflate* the gap's apparent prevalence and severity, so the quantitative pass rates should not be read as ecosystem-wide estimates. A randomly sampled rerun over the APIs.guru population would balance provider representation; this remedy is left to future evaluation rounds. The operational evaluation was conducted on a single IDP (Novelcore's KubeCore platform) with one shared cluster, so multi-cluster operational dynamics and the cross-platform portability of XSDLC remain untested.

Construct validity. SDT's principle-based scoring treats each principle as a binary pass/fail. A specification that fails P002 with one missing description and one that fails P002 with 200 missing descriptions receive the same result. A risk-based scoring model—each failed check weighted by its operational risk, yielding a continuous quality score suitable for trend analysis and graduated promotion thresholds—is recorded as future work in Chapter 11, Section 11.5. Two further concerns limit what the measured

detection rates measure. First, P006 cannot provide complete oversight of specification *drift*: it detects structural discrepancies between response and schema, but not content discrepancies, so a drifted implementation that preserves response shape goes undetected; complete drift oversight would require a content-aware oracle. Second, non-functional testing (P007) is deliberately restrained to vegeta-driven load tests against P95 latency and success-rate thresholds, leaving the broader non-functional-requirement space (throughput, resource consumption, soak and spike behaviour, concurrency limits) untested. The reported detection rates are therefore biased upward relative to a broader functional and NFR evaluation.

Statistical conclusion validity. The evaluation reports purely descriptive pass/fail counts: no confidence intervals, no significance tests, and small controlled samples (fifteen check-level defects, five evaluation APIs, three defect combinations). The claims are demonstrative—the framework detects the evaluated defect classes under the evaluated conditions—rather than statistically generalisable estimates of detection performance.

10.5 Implications for Practice

For API teams. SDT provides a concrete, automated quality standard that API teams can adopt incrementally. Starting with minimal mode (P001 only) in CI and progressing to strict mode for release pipelines allows teams to improve specification quality without requiring immediate perfection. Adoption also reduces specification drift: the runtime layer (P006) catches structural divergence from the published contract on every promotion, though this coverage is structural, not content-aware (Section 10.4). The structured validation report identifies specific gaps, enabling targeted remediation rather than blanket documentation mandates.

For platform teams. XSDLC demonstrates that quality-gated delivery pipelines can be provisioned declaratively as platform capabilities. A platform team can install the Helm chart once and expose `XSDLC` as a self-service resource that application teams create without platform team involvement. This follows the Team Topologies [64] model: the platform provides the capability; the application team exercises it.

For compliance and governance. The PR-based promotion model produces an inherent audit trail without requiring a separate compliance system. Every promotion is a pull request; every gate verdict is a commit status; every validation report is a PR comment with per-principle evidence; every execution trace is an archived Argo Workflow. For production environments with `autoMerge: false`, the human merge decision is recorded alongside

the automated assessment—satisfying separation-of-duties requirements. This auditability is a structural consequence of Git as the source of truth, not a bolted-on capability.

For the AI-assisted development community. The CLAUDE.md knowledge protocol and the persistent memory system provide a replicable pattern for structuring agent context in complex, multi-component projects. The key insight is that agent effectiveness scales with the quality of the knowledge infrastructure, not with the capability of the model alone.

10.6 Summary

The four research questions are answered affirmatively, each with a scoped qualification: SDT automates the systematic aspects of API review but not exploratory testing (RQ1), derives tests from the specification but gating policy from a second artefact (RQ2), replaces manual gate provisioning at proof-of-concept scale (RQ3), and yields an ontological specification demonstrated but not yet exercised in production (RQ4). The systemic specification quality gap and the BYOCI boundary delimit what this automation can and cannot assure, and the threats to validity follow directly from these design choices. Chapter 11 synthesizes these findings into contributions, limitations, and future work.

Chapter 11

Conclusion

11.1 Summary of Contributions

This thesis presented six principal contributions to the field of automated API quality assurance, summarised below in the same order as Section 1.6 of Chapter 1.

A conceptual framework for specification-driven API validation. Specification-Driven Testing formalises specification-driven validation as three axioms (Completeness, Determinism, Observability) operationalised by nine automatable principles (P001–P009), all derived from the OpenAPI specification with zero manual test authorship (Chapter 3). We claim it as a conceptual framework rather than a complete software-testing methodology in its own right; the elements that would elevate it to one (test derivation procedure, execution and failure semantics, lifecycle governance) are scoped as future work in Section 11.5.

The DriveBy framework. A Go CLI tool implementing SDT (Chapter 5): the `APISpec` abstraction enables version-agnostic validation across OpenAPI 3.x and Swagger 2.0; seven principle checkers apply each check only in the modes for which it is suitable, across four validation modes (minimal, strict, test-ready, test-only); and structured reports carry per-principle pass/fail status, severity classifications, and suggested remediations.

The XSDLC delivery pipeline. A Crossplane custom resource definition that transforms the DriveBy CLI from a standalone tool into a declarative, Kubernetes-native quality gate system: a single XSDLC custom resource ($\sim$35 lines of YAML) generates approximately 45–50 managed Kubernetes objects in under 15 seconds (Chapter 6). The XSDLC follows the BYOCI (Bring Your Own CI) principle: it owns delivery while leaving application CI to the developer's chosen tooling (Chapter 7).

XSDLC as ontological specification. The XSDLC CRD functions as a Kubernetes-native ontological artefact that operators and autonomous agents can discover, inspect, and manage as a composable building block within an internal developer platform. This contribution reframes quality-gated delivery as a platform capability expressed through structured, machine-readable specifications rather than imperative tooling.

Empirical evaluation. A four-arm evaluation of SDT—a controlled experiment combining check-level defect injection on a hand-crafted reference API with a purpose-built five-API defect suite, an agent-driven large-scale study over public OpenAPI specifications from APIs.guru, a proof-of-concept deployment on the Novelcore IDP, and an agent-feedback experiment in which an AI agent, using the DriveBy report as its sole feedback signal, improved the Swagger Petstore specification from 1/6 to 4/6 passing principles (Section 8.6)—establishes that the framework detects real specification-quality gaps, integrates into a live GitOps pipeline, operates without dedicated testers in the promotion path, and produces feedback actionable by autonomous agents (Chapter 8).

An agent-driven development framework. The CLAUDE.md knowledge protocol—a hierarchical system of sixteen machine-first documentation files—provided AI agents with persistent, structured context during the 12-day Phase B partnership window described in Chapter 9, in which the author and the agent jointly produced (under the author's direction and review) the productisation polish, the multi-API evaluation pipeline, and the thesis-writing scaffolding. We claim this as an enabling framework within an existing software development lifecycle, not a standalone methodology. SDT's structured output proved directly consumable by AI agents, validating the Axiom of Observability's practical utility beyond CI/CD gating.

11.2 Answers to Research Questions

RQ1: *Can a conceptual framework grounded in specification completeness, behavioural determinism, and operational observability demonstrate the feasibility of automated API quality assurance that can operate without dedicated testers in the promotion pipeline?*

Yes. The controlled evaluation (Section 8.3) detected all seven principle-level injected defects with zero false positives; at the finer-grained check level, the per-check extension (Section 8.3.4) detected 12 of 15 single-check defects (80%)—of the three non-detections, one was neutralised by the loader's structural validation, one proved an injection artefact with no real defect, and exactly one traces to a known framework limitation—and all three evaluable

multi-principle combinations. The large-scale evaluation (Section 8.5) across real-world APIs from the APIs.guru registry confirmed that SDT detects real quality gaps that even canonical specifications (Swagger Petstore v3) exhibit. The three axioms are not aspirational—they are necessary conditions for the validation loop to function: Completeness ensures the specification is a sufficient test oracle, Determinism ensures reproducible gate decisions, and Observability ensures actionable feedback for CI/CD pipelines and AI agents alike, a sufficiency the agent-feedback experiment (Section 8.6) demonstrated empirically.

RQ2: *To what extent can validation rules, test cases, and quality gates be derived automatically from the OpenAPI specification together with declared operational policy (the XSDLC resource), and does the resulting automation produce actionable, machine-readable signals suitable for both human engineers and autonomous agents?*

Fully, for the implemented static principles. All seven principle checkers derive their validation logic entirely from the OpenAPI specification. No test cases are authored by hand; no test data is manually curated. The qualification: performance gating (P007) is parameterised by thresholds declared in the XSDLC custom resource rather than derived from the OpenAPI document, so the zero-authorship property holds for the static principles and functional testing, while non-functional thresholds remain a declared deployment-policy artefact (Chapter 2). The test-ready mode (P009) explicitly validates whether the specification provides sufficient detail for automated test generation. The multi-API controlled evaluation (Section 8.4) showed each defect API blocked by exactly the principle matching its injected defect class, and the large-scale evaluation confirmed meaningful validation results for any valid OpenAPI specification without per-API configuration.

RQ3: *How effectively does specification-driven quality assurance integrate into cloud-native GitOps pipelines, and can a single declarative custom resource replace the manual provisioning of quality gate infrastructure?*

Effectively, as demonstrated by the proof-of-concept deployment on Novelcore's cluster. A single XSDLC custom resource (∼35 lines of YAML) generates approximately 45–50 managed Kubernetes objects in under 15 seconds, replacing hours of manual provisioning across the ArgoCD UI, GitHub UI, and `kubectl`. The CI/CD integration metrics (Section 8.2.8) point the same way: sub-second CLI startup, 2–5 second strict-mode validation, and under 90 seconds of end-to-end gate latency from webhook to posted commit status keep the gate inside interactive developer feedback loops. The integration also scaled: the multi-API evaluation (Section 8.4) ran five concurrent XSDLC-provisioned pipelines without operational problems, and the surgical coexistence model allowed installation into—and removal from—an existing

cluster without modifying its running ArgoCD, Argo Workflows, and Argo Events installations. Every promotion was automatically validated against both the specification contract and configurable performance thresholds, demonstrating the feasibility of operating quality gates without dedicated testers in the promotion pipeline.

RQ4: *Can quality-gated delivery pipelines be expressed as Kubernetes-native ontological specifications that operators and autonomous agents can discover, inspect, and manage as composable building blocks within an internal developer platform?*

Yes. Section 11.3 demonstrates that the XSDLC CRD functions as a machine-readable, structurally typed ontological artefact: in Novelcore's IDP it is one Crossplane composition among many, discoverable via the Kubernetes API, inspectable, and manageable by higher-level operators as a child resource. Its structural typing means that a team onboarding controller could create XSDLC resources without understanding the composition's internals, and an observability controller could adjust load testing thresholds without human intervention; these controllers are not yet implemented (see Section 11.5).

11.3 XSDLC as Ontological Specification

The most significant architectural insight of this thesis is that XSDLC is not a tool—it is a *specification*. Specifically, it is a Kubernetes Custom Resource Definition that describes a quality-gated delivery pipeline as a machine-readable, structurally typed, discoverable ontological artefact [37].

This distinction matters because it determines how the system composes with other systems. A tool must be invoked; a specification can be *discovered*. Any Kubernetes controller, operator, or autonomous agent can:

- **Discover** XSDLC resources via the Kubernetes API (`kubectl get xsdlcs`).
- **Inspect** their declared delivery pipeline—environments, quality gates, thresholds.
- **Manage** them as child resources, e.g. a team onboarding controller creating an XSDLC while provisioning a new microservice.
- **Update** individual fields. An observability controller could adjust `loadTestConfig.concurrentUsers` to production traffic patterns; a security policy controller could enforce that every XSDLC includes a `validate-only` check with `validationMode: strict`, guaranteeing

the security principle is always evaluated—though the coverage it enforces is limited to what P005 checks, the presence and operation-level wiring of declared authentication schemes; deeper security testing is future work (Section 11.5).

In Novelcore's IDP, XSDLC sits alongside database, networking, and observability compositions, all following the same pattern: a custom resource declares intent, a composition generates implementation, and controllers reconcile toward the declared state. The XSDLC's value here is not its validation logic (which the DriveBy CLI provides) but its *structural integration* into the Kubernetes resource model: typed, versioned, discoverable, reconcilable.

This framing connects to the platform engineering movement [40, 64]: a CRD is the Kubernetes-native *self-service interface*, exposing a simplified API surface (the XRD schema) that abstracts away implementation complexity (the composition). The XSDLC demonstrates that quality-gated delivery can be delivered through this same interface, making quality assurance a platform capability rather than a per-team responsibility.

The transition from CLI flags to Custom Resource Definition (Section 4.3) is a change not of tool but of *paradigm*: the operator stops telling the system what to do and declares how the world should *be*, and the control plane converges toward that state continuously. This shift—from imperative instruction to ontological declaration—is the central architectural insight of this thesis, and it applies far beyond API validation.

11.4 Limitations

The limitations below were identified through three channels: the empirical evaluation (Chapter 8), critical examination of the framework's conceptual scope, and deliberate implementation scope decisions.

Verdict tier is open-loop. The controller in the verdict tier (Section 7.6.2) decides *promote* or *block*, but does not itself emit a corrective signal u that reduces the error $e(t)$ on the plant. There is no automated spec re-write, no implementation patch, no autonomous remediation. The error signal is exported as a structured report; the corrective action is performed externally—by a developer or, in Section 8.6, by an AI coding agent consuming the same report. This is the deliberate, scoped open-loop component of an otherwise self-reconciling system, and closing it autonomously is recorded as future work below. Drift between gate evaluations is *not* a separate limitation: the promotion chain actively blocks any production-bound transition that has not first passed P006, so spec–implementation divergence cannot reach production by construction (Section 7.6.2).

Principle coverage. Two of nine principles (P006: Functional Testing, P007: Performance Testing) are defined theoretically but not yet implemented as `PrincipleChecker` wrappers. The underlying testing logic exists in the CLI but does not contribute to the structured validation report. Closing this gap must preserve the layered-validation design: strict mode remains *static-only*, so the P006/P007 wrappers would join the dynamic `test-only` mode (or a composed full-validation mode; Section 11.5), never the static modes. P006 itself has a depth limit: it verifies that responses *structurally* comply with the declared schemas (status codes, types, required fields), not that the response *content* is semantically correct for the request.

Keyword-based checks. Several P008 checks (versioning strategy, breaking changes, migration guides) rely on keyword matching in the `info.description` field. This approach is brittle: a specification that documents its versioning strategy in a separate markdown file rather than in the OpenAPI description would fail P008 despite having adequate documentation. More fundamentally, P008 verifies only that a versioning strategy is *documented*, not how well versioning is *supported* by the API—assessing the latter would require understanding the semantics of the applied strategy (URI versioning, header negotiation, deprecation windows) and testing the API's behaviour against them.

Single specification format. DriveBy currently supports OpenAPI 3.x and Swagger 2.0. APIs documented in other formats (GraphQL schemas, gRPC protobuf definitions, AsyncAPI specifications) are not covered. The `APISpec` abstraction layer is designed to accommodate additional adapters, but none have been implemented.

Load testing scope. The current load testing implementation validates against static thresholds declared in the XSDLC spec. It does not account for time-varying traffic patterns, seasonal load spikes, or progressive capacity growth. The thresholds represent a point-in-time business requirement, not an adaptive contract. Load testing is also the *only* non-functional testing performed: stress and soak testing, security testing beyond P005's authentication-scheme checks, and reliability/chaos testing are not covered.

Evaluation scale. The large-scale evaluation uses public APIs from APIs.guru, which may not be representative of internal enterprise APIs. Internal APIs often have different documentation practices, security models, and quality standards. The operational evaluation is limited to a single IDP (Novelcore) and one shared cluster, albeit exercising five concurrent XSDLC-provisioned pipelines.

Single delivery cluster. The XSDLC composition is exercised against one ArgoCD/Crossplane cluster (`private.novelcore.org`); multi-cluster promotion, where environments span distinct Kubernetes clusters, is not

exercised. The architectural shape supports it—ArgoCD ApplicationSets and multi-cluster Crossplane providers exist—but the present evaluation does not test the cross-cluster orchestration path.

11.5 Future Work

Each direction below addresses a specific limitation from Section 11.4.

Adaptive load testing thresholds. A Kubernetes controller that watches production observability metrics (Prometheus, Grafana) and adjusts `loadTestConfig` fields to actual traffic patterns, tightening the gate to the observed operational reality. This would partially close the observability–gating feedback loop—partially, because it improves only the gate thresholds in the XSDLC resource, not the OpenAPI specification itself, which may also need improvement.

Risk-based scoring model. A severity-weighted scoring model in which each failed check contributes a weight proportional to its operational risk—a missing security scheme (P005) outweighs a missing example (P002). The implementation deliberately keeps a binary verdict for gating simplicity; a risk-based score would complement it with a continuous quality signal for trend dashboards, graduated promotion thresholds, and longitudinal comparison. The conceptual work is calibrating the weights: expert judgement, the API-quality literature, or learned remediation effort. This is the natural follow-up to the construct-validity discussion in Chapter 10.

Agent-driven remediation. Autonomous engineering agents [77, 23] that parse SDT validation reports, identify failing principles, generate fixes, and re-push to trigger re-validation. The scope is deliberately *specification-first*: agents remediate the OpenAPI specification (or, for gate-configuration failures, the XSDLC resource), keeping the implementation as the stable ground truth; extending remediation to application code is explicitly conditional on first establishing a reliable procedure for deciding which artefact—specification or implementation—is authoritative when the two disagree. The structured output required by the Axiom of Observability is precisely the interface agents need, as the agent-feedback experiment (Section 8.6) demonstrated for a single report-driven pass. The target, however, is a multi-round validate–remediate–revalidate loop converging to complete mitigation of all principle failures; characterising that convergence at scale is the doctoral extension recorded in Section 11.5.1.

P006 and P007 as PrincipleCheckers. Wrapping the existing functional and performance testing logic into the unified principle evaluation pipeline, so runtime results contribute to the structured report in the same per-

principle format as static checks. The extension preserves the static/dynamic layering: strict mode remains static-only, permanently, and the wrappers join the `test-only` mode; a complete nine-principle assessment would come from a composed *full validation* run that executes both layers and merges their reports, never from folding runtime testing into a static mode.

Advanced security testing. Security coverage today is what P005 checks: declared security schemes and their operation-level wiring. A near-term extension is dynamic security-testing scenarios derived from the specification—verifying that protected operations reject unauthenticated and under-scoped requests and that undeclared endpoints are unreachable. Like P006/P007, these would live in the dynamic validation layer, not the static modes.

Performance thresholds as an OpenAPI vendor extension. A vendor extension (e.g. `x-performance`) carrying non-functional thresholds—latency percentiles, throughput, concurrent-user envelopes—inside the OpenAPI document itself. The performance gate would then be derived from the specification rather than parameterised by the XSDLC resource, removing the single qualification on the zero-authorship claim (Chapter 2); the XSDLC CR would retain only gate *policy*. This is the near-term form of the metamodel-level extension recorded in Section 11.5.1.

Scheduled prod-environment gate. The per-PR gate guarantees that no spec–implementation divergence introduced through the pipeline reaches production (Section 7.6.2). A complementary engineering extension—not a research direction—is a CronWorkflow trigger executing P006 over the deployed prod API on a schedule, reusing the same XSDLC gate machinery. This covers the residual class of changes that bypass the promotion chain: out-of-band operator intervention, infrastructure-level shifts (runtime upgrades, library auto-updates), or data-driven divergences that do not manifest at PR time.

Multi-format specification support. Extending the `APISpec` abstraction layer with adapters for GraphQL introspection schemas, gRPC protobuf definitions, and AsyncAPI specifications. The extension is more than an interface exercise: those formats lack first-class equivalents for documented error surfaces, security-scheme wiring, examples, and versioning narratives, and must be extended or supplemented before the checkers can produce comparable verdicts.

Enterprise multi-cluster patterns. Extending XSDLC to support multi-cluster delivery pipelines where environments span different Kubernetes clusters (e.g. development on a shared cluster, production on a dedicated cluster). This would require cross-cluster ArgoCD ApplicationSets [9] and multi-cluster Crossplane providers [26].

XSDLC as operator child resource. Realising the composition pattern of Section 11.3 at scale: a microservice provisioning operator declares an XSDLC resource as a child alongside database credentials, networking policies, and observability configurations, making quality assurance a declared attribute of every service rather than a separate concern.

11.5.1 Doctoral Continuation

Several research directions exceed the scope of a diploma thesis but constitute a coherent doctoral programme building directly on the foundations laid here.

From conceptual framework to formal methodology. The most consequential extension is to upgrade SDT from a conceptual framework—three axioms, nine principles, an empirical implementation—to a complete software-testing methodology. This requires three additions: (i) a formal *test derivation procedure* that maps from the specification to a test suite (controlled by a derivation logic, with completeness and minimality theorems); (ii) explicit *execution and failure semantics* (a labelled transition system over validation states, with axioms over how partial failures compose into a global verdict); and (iii) *lifecycle governance* (when a principle is added, deprecated, refined, or split; how methodology versions relate). Together they would make SDT a peer to TDD, BDD, and Domain-Driven Design rather than a candidate for that role.

Coverage extension across testing types. The framework covers contract validation (P001–P005, P008, P009), functional testing (P006), and load testing (P007), but a complete API testing programme also includes unit, integration, end-to-end, security, and reliability/chaos testing. The conceptual question is whether each can be expressed as a principle (or family of principles) over the specification, or requires complementary frameworks; the empirical question is what the principle space looks like at full coverage.

Stress testing as a first-class concern. The load-testing principle (P007) tests the system under *expected* load. A complementary stress-testing principle would test the system under *breaking* load, surfacing the operational envelope rather than the operational baseline. The methodological question is how to derive stress-test parameters from the specification: declared rate-limit annotations, declared SLA fields, observed historical load patterns.

Ontological extension to performance and security metrics. OpenAPI's vocabulary lacks first-class concepts for performance characteristics (expected latency, throughput, percentile guarantees) and for security-testing semantics beyond declared authentication schemes (threat assumptions, enforcement behaviours, scope semantics). Extending the OpenAPI metamodel with these annotation vocabularies, then deriving the corresponding testing

principles from them, would let the contract specify both the ideal functional *and* non-functional behaviour of the API, leaving the XSDLC CR to specify mainly gate policies.

Conditional gate transitions. A *conditional* gate would let a build through subject to constraints (e.g. "promote to staging only if observed P95 latency in the last 24h is below 100 ms"), requiring a formal language for single and composite gate conditions, an evaluator integrated with cluster observability, and an audit trail for promotion decisions. The delta is precise: today's gates evaluate point-in-time load-test results against XSDLC-declared thresholds; conditional gates would evaluate constraints over live, time-windowed production observability data.

Outer loop with dynamic setpoint over the existing two-tier stack. The doctoral extension to the two-tier controller stack (Section 7.6.2) is a *third, outer* loop that adjusts the verdict-tier setpoint based on observed system behaviour: an SLA-based controller that tightens the quality threshold when observed latency creeps up and loosens it when stability is established. This is control-theoretic SDT—the natural research bridge between specification-driven testing and self-adjusting platforms.

Empirical proof of agent-driven correction at scale. The single-spec, single-iteration experiment in Section 8.6 establishes that the SDT report is sufficient feedback for an agent to make measurable corrections. The doctoral extension is the multi-spec, multi-iteration study—dozens of public APIs, multiple agents, multiple revision passes, with statistical analysis of convergence rates, failure modes, and the agent characteristics that predict success—converting "feedback infrastructure for agent-driven development" from a substantiated claim into a fully characterised empirical regime.

These directions are not independent: the formal methodology is the substrate on which the others rest, and a doctoral plan would address them roughly in the order listed, with the formalisation programme spanning the first eighteen months.

The thesis closes where it began: with the specification. SDT's central claim—that a sufficiently complete, deterministic, and observable API specification can carry the full weight of automated quality assurance—has survived contact with a reference implementation, a live GitOps platform, seventy public specifications, and an autonomous coding agent. The framework presented here is deliberately a foundation rather than a finished edifice: its axioms are stated so that they can be formalised, its principles so that they can be extended, and its gates so that they can be governed. If the GitOps era has taught infrastructure engineering that the declared state can be the operational truth, this thesis argues the same lesson for API quality—the specification is not documentation about the system; it is the system's contract,

and it can be made to enforce itself.

Bibliography

[1] *42Crunch: API Security Audit and Conformance Scanning*. https://42crunch.com/. Continuous OpenAPI security auditing and runtime conformance enforcement. 2024.

[2] Anthropic. *Claude Code: An Agentic Coding Tool*. https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview. Accessed: 2026-03-01. 2025.

[3] Apiary. *Dredd: HTTP API Testing Framework*. https://dredd.org. Now maintained by the open-source community. 2024.

[4] *APIClarity: Reconstructing OpenAPI Specifications from API Traffic*. https://github.com/openclarity/apiclarity. CNCF Sandbox project for traffic-derived API specification capture. 2024.

[5] Apideck. *Portman: Port OpenAPI Specs to Postman Collections*. https://github.com/apideck-libraries/portman. 2024.

[6] APIs.guru. *APIs.guru: Wikipedia for Web APIs*. https://apis.guru. 2024.

[7] Andrea Arcuri. "EvoMaster: Evolutionary Multi-context Automated System Test Generation". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.

[8] Argo CD. *Argo CD Source Hydrator*. https://argo-cd.readthedocs.io/en/latest/user-guide/source-hydrator/. Declarative manifest rendering via Kustomize/Helm with hydrated output branches. 2025.

[9] *Argo CD - Declarative GitOps CD for Kubernetes*. https://argoproj.github.io/cd/. 2024.

[10] *Argo Events - Event-driven Workflow Automation*. https://argoproj.github.io/events/. 2024.

[11] *Argo Workflows - Container-native Workflow Engine*. https://argoproj.github.io/workflows/. 2024.

[12] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. 2nd. Princeton University Press, 2021. ISBN: 978-0691213477.

[13] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. "Checking Security Properties of Cloud Service REST APIs". In: *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020, pp. 387–397.

[14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. "RESTler: Stateful REST API Fuzzing". In: *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2019, pp. 748–758.

[15] Carlos Alexandro Becker. *GoReleaser: Release Automation for Go Projects*. `https://goreleaser.com`. 2024.

[16] Brendan Burns et al. *Kubernetes: Up and Running*. 3rd. O'Reilly Media, 2022. ISBN: 978-1098110208.

[17] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2000, pp. 268–279.

[18] Cloud Native Computing Foundation. *CNCF Annual Survey 2023*. `https://www.cncf.io/reports/cncf-annual-survey-2023/`. Accessed: 2025-12-01. 2023.

[19] Cloud Native Computing Foundation. *CNCF Platforms White Paper*. `https://tag-app-delivery.cncf.io/whitepapers/platforms/`. Accessed: 2025-12-01. 2023.

[20] Cloud Native Computing Foundation. *Flux: The GitOps Family of Projects*. `https://fluxcd.io`. 2024.

[21] Cloud Native Computing Foundation. *Open Policy Agent: Policy-Based Control for Cloud Native Environments*. `https://www.openpolicyagent.org`. 2024.

[22] CNCF OpenGitOps Working Group. *OpenGitOps: Principles for GitOps*. `https://opengitops.dev`. Version 1.0.0. 2022.

[23] Cognition AI. *Devin: The First AI Software Engineer*. `https://www.cognition.ai/blog/introducing-devin`. Accessed: 2025-12-01. 2024.

[24] Melvin E. Conway. "How Do Committees Invent?" In: *Datamation* 14.4 (1968), pp. 28–31.

[25] Crossplane. *Crossplane Compositions*. https://docs.crossplane.io/latest/concepts/compositions/. Pipeline-mode composition with functions. 2024.

[26] *Crossplane: The Cloud Native Control Plane*. https://crossplane.io. 2024.

[27] Docker. *Docker Buildx: Extended Build Capabilities*. https://docs.docker.com/buildx/working-with-buildx/. 2024.

[28] Dmitrii Dygalo. *Schemathesis: Property-Based Testing for API Schemas*. https://schemathesis.readthedocs.io. 2024.

[29] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018. ISBN: 978-1942788331.

[30] Steve Francia. *Cobra: A Commander for Modern Go CLI Interactions*. https://cobra.dev. 2024.

[31] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 978-0201633610.

[32] getkin. *kin-openapi: OpenAPI 3.x Implementation for Go*. https://github.com/getkin/kin-openapi. 2024.

[33] GitHub. *GitHub Actions: Automate Your Workflow*. https://github.com/features/actions. 2024.

[34] *GitOps Promoter for Kubernetes*. https://github.com/argoproj-labs/gitops-promoter. 2024.

[35] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. "Testing RESTful APIs: A Survey". In: *ACM Transactions on Software Engineering and Methodology* 33.1 (2023), pp. 1–41.

[36] Google Cloud. *DORA Metrics: The Four Keys to Software Delivery Performance*. https://dora.dev. 2024.

[37] Thomas R. Gruber. "A Translation Approach to Portable Ontology Specifications". In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220.

[38] HashiCorp. *Terraform: Infrastructure as Code*. https://www.terraform.io. 2024.

[39] Helm. *Helm OCI Support: Registries*. https://helm.sh/docs/topics/registries/. OCI-compliant artifact storage for Helm charts. 2024.

[40] Humanitec. *State of Platform Engineering Report*. https://humanitec.com/whitepapers/state-of-platform-engineering. Accessed: 2025-12-01. 2024.

[41] JSON Schema. *JSON Schema Specification 2020-12*. https://json-schema.org/specification. Accessed: 2025-12-01. 2022.

[42] *Kong Gateway: Cloud-Native API Gateway*. https://konghq.com/products/kong-gateway. API gateway enforcing specification-derived policies at the edge. 2024.

[43] Kubernetes. *Kubernetes API Conventions*. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md. Spec/status pattern and controller conventions. 2024.

[44] Kubernetes. *Kubernetes Controllers*. https://kubernetes.io/docs/concepts/architecture/controller/. Observe-diff-act reconciliation pattern. 2024.

[45] *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io. 2024.

[46] Kyverno. *Kyverno: Kubernetes Native Policy Management*. https://kyverno.io. 2024.

[47] Microcks. *Microcks: Open-Source Cloud-Native Tool for API Mocking and Testing*. https://microcks.io. CNCF Sandbox project. 2024.

[48] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, 1995. ISBN: 978-0195092691.

[49] OpenAPI Initiative. *About the OpenAPI Initiative*. https://www.openapis.org/about. History of the specification from Swagger to OpenAPI. 2024.

[50] OpenAPI Initiative. *Swagger Petstore: Sample OpenAPI 3.0 Definition*. https://petstore3.swagger.io. 2024.

[51] *OpenAPI Specification*. https://spec.openapis.org/oas/v3.1.0. 2024.

[52] Optic. *Optic: Understand Your API Changes*. https://www.useoptic.com. 2024.

[53] OWASP. *OWASP API Security Top 10 — 2023*. https://owasp.org/API-Security/. Accessed: 2025-12-01. 2023.

[54] Pact Foundation. *Pact: Contract Testing for Microservices*. https://pact.io. 2024.

[55] pb33f. *Vacuum: The World's Fastest OpenAPI Linter*. https://quobix.com/vacuum/. 2024.

[56] Sida Peng et al. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. arXiv preprint arXiv:2302.06590. 2023.

[57] Postman. *2023 State of the API Report*. https://www.postman.com/state-of-api/. Accessed: 2025-12-01. 2023.

[58] Tom Preston-Werner. *Semantic Versioning 2.0.0*. https://semver.org. Accessed: 2025-12-01. 2013.

[59] Pulumi. *Pulumi: Infrastructure as Code in Any Programming Language*. https://www.pulumi.com. 2024.

[60] Sebastián Ramírez. *FastAPI: Modern, Fast Web Framework for Building APIs with Python*. https://fastapi.tiangolo.com. 2024.

[61] Redocly. *Redocly CLI: OpenAPI Linting and Bundling*. https://redocly.com/docs/cli. 2024.

[62] Ian Robinson. *Consumer-Driven Contracts: A Service Evolution Pattern*. https://martinfowler.com/articles/consumerDrivenContracts.html. Published on martinfowler.com. 2006.

[63] Tomás Senart. *Vegeta: HTTP Load Testing Tool and Library*. https://github.com/tsenart/vegeta. 2024.

[64] Matthew Skelton and Manuel Pais. *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press, 2019. ISBN: 978-1942788812.

[65] Antonios Smardas and Kyriakos Kritikos. "Towards the Automatic Production of OpenAPI Specifications from Source Code". In: *12th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2025, pp. 342–349.

[66] SmartBear. *SwaggerHub: Design, Document, and Build APIs*. https://swagger.io/tools/swaggerhub/. 2024.

[67] SmartBear. *The State of API Report 2023*. https://smartbear.com/state-of-software-quality/api/. Accessed: 2025-12-01. 2023.

[68] Larry Smith. "Shift-Left Testing". In: *Dr. Dobb's Journal* 26.9 (2001). Popularized the concept of moving testing earlier in the development lifecycle.

[69] springdoc. *springdoc-openapi: Library for OpenAPI 3 with Spring Boot*. https://springdoc.org. 2024.

[70] Stack Overflow. *2024 Developer Survey*. https://survey.stackoverflow.co/2024/. Accessed: 2025-12-01. 2024.

[71] Stoplight. *Spectral: Open-Source API Style Guide Enforcer and Linter*. `https://stoplight.io/open-source/spectral`. 2024.

[72] Stoplight. *Stoplight: API Design, Documentation, and Governance Platform*. `https://stoplight.io`. 2024.

[73] Synadia Communications. *NATS JetStream: Persistence for NATS*. `https://docs.nats.io/nats-concepts/jetstream`. 2024.

[74] Petros Evangelos Triantafyllis. *DriveBy: Specification-Driven Testing CLI*. `https://github.com/meter-peter/driveby`. 2025.

[75] VMware. *Spring Cloud Contract*. `https://spring.io/projects/spring-cloud-contract`. 2024.

[76] Lei Wang et al. "A Survey on Large Language Model Based Autonomous Agents". In: *Frontiers of Computer Science* 18.6 (2024), p. 186345.

[77] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.