National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

A Comparative Study of Model-First API Design Frameworks and a Methodology for Endpoint Interdependency Analysis

Diploma Thesis

Charalampos Kampougeris

Supervisor: Vassilios Vescoukis
Professor NTUA

Athens, November, 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

A Comparative Study of Model-First API Design Frameworks and a Methodology for Endpoint Interdependency Analysis

Diploma Thesis

Charalampos Kampougeris

Supervisor: Vassilios Vescoukis
Professor NTUA

Approved by the three-member scientific committee on November 5th, 2025.

.....	.....	.....
Vassilios Vescoukis	Nikolaos Papaspyrou	Konstantinos Sagonas
Professor NTUA	Professor NTUA	Associate Professor NTUA

Athens, November, 2025

.....

Charalampos Kampougeris
Graduate of School of Electrical and Computer Engineering, National Technical University of Athens

Περίληψη

Στην εποχή όπου τα πληροφοριακά συστήματα αποτελούν τη ραχοκοκαλιά της ψηφιακής υποδομής, τα Application Programming Interfaces (APIs) διαδραματίζουν κρίσιμο ρόλο στη διαλειτουργικότητα μεταξύ υπηρεσιών, εφαρμογών και χρηστών. Τα APIs ορίζουν συμβόλαια με σαφή δομή, σύνολα από endpoints που εκθέτουν συγκεκριμένες λειτουργίες ενός συστήματος. Καθώς οι σύγχρονες αρχιτεκτονικές βασίζονται όλο και περισσότερο σε κατανεμημένες και cloud-native υπηρεσίες, η συνέπεια, η επεκτασιμότητα και η συντηρησιμότητα των APIs είναι καθοριστικής σημασίας για τον αποδοτικό σχεδιασμό και την εξέλιξη συστημάτων. Η παρούσα εργασία επικεντρώνεται στη μεθοδολογία Model-First API Design, όπου το συμβόλαιο του API ορίζεται πριν την υλοποίηση, με χρήση εργαλείων. Τρία αντιπροσωπευτικά frameworks που μελετώνται είναι τα gRPC/Protocol Buffers, AWS Smithy και Microsoft TypeSpec. Κάθε framework παρέχει έναν δομημένο τρόπο περιγραφής endpoints, λειτουργιών και δεδομένων, υποστηρίζοντας αυτόματη παραγωγή τεκμηρίωσης και SDKs. Με βάση αυτή τη θεμελίωση, η εργασία προτείνει μια μεθοδολογία για στατική ανάλυση εξαρτήσεων μεταξύ endpoints, αξιοποιώντας τα μοντέλα που παράγονται από τα παραπάνω frameworks. Η ανάλυση εντοπίζει εξαρτήσεις δεδομένων μεταξύ των endpoints συγκρίνοντας κοινούς τύπους και δομές στα εισερχόμενα και εξερχόμενα μοντέλα. Αυτό επιτρέπει την οπτικοποίηση των σχέσεων μεταξύ λειτουργιών και την αποκάλυψη έμμεσων ακολουθιών κλήσεων που αντανακλούν πραγματικές ροές API. Τα πειραματικά αποτελέσματα δείχνουν ότι τα μοντέλα Smithy και TypeSpec παράγουν ιδιαίτερα ακριβή και καθαρά dependency graphs, εντοπίζοντας εξαρτήσεις με μειωμένη αμφισημία. Αντιθέτως, το gRPC παρουσιάζει χαμηλότερη ακρίβεια λόγω της αδυναμίας του να ορίζει wrapped primitive types, κάτι που οδηγεί σε απώλεια σημασιολογικής πληροφορίας κατά την εξαγωγή σε OpenAPI. Τα ευρήματα καταδεικνύουν ότι οι καλά δομημένοι, model-first ορισμοί προσφέρουν πιο αξιόπιστη βάση για ανάλυση εξαρτήσεων APIs σε σχέση με παραδοσιακές στατικές ή δυναμικές προσεγγίσεις όπως εκείνες που εφαρμόζονται στο RADAR.

Λέξεις Κλειδιά: Model-First API Σχεδιασμός, gRPC, Smithy, TypeSpec, Ανάλυση Εξαρτήσεων API, OpenAPI, Smithy AST, Στατική Ανάλυση

Abstract

In an era where software systems form the backbone of digital infrastructure, Application Programming Interfaces (APIs) play a crucial role in enabling interoperability between services, applications, and users. APIs define structured contracts, sets of endpoints that expose specific functionalities of a system. As modern architectures increasingly rely on distributed and cloud-native services, the consistency, scalability, and maintainability of APIs have become vital for efficient system design and evolution. This thesis focuses on the Model-First API Design approach, where the API contract is defined before implementation, using formal modeling frameworks. Three such frameworks are examined: gRPC/Protocol Buffers**,** AWS Smithy, and Microsoft TypeSpec. Each framework provides a structured way to describe endpoints, operations, and data models, supporting automated documentation and SDK generation. Building on this foundation, the thesis introduces a methodology for static inter-endpoint dependency analysis that leverages the structured models produced by these frameworks. The analysis identifies data dependencies between API endpoints by comparing shared types and message structures in their input and output models. This enables the visualization of relationships between operations and the discovery of implicit invocation orders that describe real API workflows. Experimental results show that Smithy and TypeSpec generate highly accurate and noise-free dependency graphs, effectively capturing object-level dependencies and reducing ambiguity. In contrast, gRPC exhibits reduced precision due to its inability to wrap primitive types, which leads to semantic loss in its OpenAPI export. The findings demonstrate that well-structured, model-first definitions provide a richer and more reliable foundation for analyzing API interdependencies than traditional static or dynamic approaches such as those implemented in RADAR.

Keywords: Model-First API Design, gRPC, Smithy, TypeSpec, API Dependency Analysis, OpenAPI, Smithy AST, Static Analysis

Acknowledgements

I would like to thank everyone who supported me throughout this journey. My sincere thanks to Professor Vassilis Vescoukis, who offered me the opportunity to work on this topic and provided continuous guidance, support, and feedback throughout the process. I would also like to thank ECE graduate Dimitris Gerokonstantis for his helpful advice during this work. Finally, I am grateful to my family for their constant support and encouragement during this period.

Table of Contents

Table of Figures

Greek Extended Abstract

1. Εισαγωγή

Στο σύγχρονο ψηφιακό οικοσύστημα, τα APIs αποτελούν τον κρίσιμο μηχανισμό διασύνδεσης συστημάτων, διαμοιρασμού δεδομένων και υλοποίησης ολοκληρωμένων υπηρεσιών. Δεν αντιμετωπίζονται πλέον ως βοηθητικά τεχνικά εργαλεία, αλλά ως στρατηγικά μέσα που καθορίζουν τον τρόπο με τον οποίο οργανισμοί εκθέτουν λειτουργίες, διαμορφώνουν οικοσυστήματα και καινοτομούν.

Η εξάπλωση κατανεμημένων αρχιτεκτονικών, όπως microservices, serverless και cloud-native εφαρμογές, έχει αυξήσει δραστικά την πολυπλοκότητα της ανάπτυξης APIs. Η ανάγκη για συνέπεια, διαλειτουργικότητα και σαφήνεια σε περιβάλλοντα με εκατοντάδες εξαρτώμενα endpoints αποτελεί μείζον πρόβλημα μηχανικής και διακυβέρνησης. Τα APIs πλέον λειτουργούν ως συμβόλαια μεταξύ ομάδων, επηρεάζοντας όχι μόνο την τεχνική υλοποίηση, αλλά και τα επιχειρησιακά μοντέλα, τις πολιτικές ασφάλειας και τις αυτοματοποιήσεις κύκλου ζωής.

Παραδοσιακά, τα περισσότερα APIs σχεδιάζονταν με τη λογική του code-first: οι μηχανικοί υλοποιούσαν τη λογική σε κώδικα και στη συνέχεια δημιουργούσαν τεκμηρίωση ή προδιαγραφές με εργαλεία όπως το OpenAPI. Αν και λειτουργική για μικρά συστήματα, αυτή η προσέγγιση οδηγεί συχνά σε κατακερματισμό: η τεκμηρίωση αποκλίνει από την υλοποίηση, ο σχεδιασμός ελέγχεται καθυστερημένα και η συνεργασία περιορίζεται σε στενά τεχνικά πλαίσια.

Ως απάντηση, οι σύγχρονοι οργανισμοί υιοθετούν την προσέγγιση model-first, όπου το API περιγράφεται πλήρως και τυπικά πριν από την υλοποίηση, μέσω γλωσσών μοντελοποίησης όπως τα Protocol Buffers (gRPC), Smithy και TypeSpec. Το μοντέλο λειτουργεί ως η μοναδική πηγή αλήθειας, από την οποία παράγονται αυτόματα κώδικας, τεκμηρίωση, SDKs και εργαλεία ελέγχου. Αυτή η πρακτική προάγει την πρώιμη επικύρωση, τη συνεπή αρχιτεκτονική και τη συνεργασία μεταξύ τεχνικών και μη τεχνικών ρόλων.

Καθώς τα APIs αναπτύσσονται πλέον σε περιβάλλοντα multi-cloud ή hybrid, η υιοθέτηση μοντέλων διευκολύνει την ανεξαρτησία από υποδομές και την αυτοματοποιημένη διακυβέρνηση του κύκλου ζωής. Η σύγκλιση του model-first σχεδιασμού με τις αρχές του DevOps και του infrastructure-as-code σηματοδοτεί μια νέα εποχή, όπου τα APIs αντιμετωπίζονται ως versioned και συνεχώς επικυρωμένα artifacts.

Η παρούσα εργασία μελετά εις βάθος αυτή την προσέγγιση. Εστιάζει στα τρία κυρίαρχα frameworks, gRPC/Protocol Buffers, AWS Smithy και Microsoft TypeSpec, και προτείνει μια μεθοδολογία για την ανάλυση εξαρτήσεων μεταξύ endpoints με στατικό τρόπο, βασισμένη σε model-first αναπαραστάσεις. Η έρευνα συνδυάζει θεωρητική ανάλυση, συγκριτική μελέτη και εμπειρική αξιολόγηση μέσω μελέτης περίπτωσης στο PayPal API, επεκτείνοντας την προηγούμενη εργασία RADAR για ανίχνευση εξαρτήσεων σε REST APIs.

2. Θεμελιώδη του Σχεδιασμού API

Τα APIs αποτελούν τον βασικό μηχανισμό διασύνδεσης και ενοποίησης σύγχρονων συστημάτων λογισμικού. Λειτουργούν ως επίσημα συμβόλαια επικοινωνίας μεταξύ πελατών και υπηρεσιών, καθορίζοντας τις διαθέσιμες λειτουργίες, τα αναμενόμενα δεδομένα εισόδου/εξόδου και τους κανόνες της αλληλεπίδρασης. Από τις πρώιμες υλοποιήσεις μέσω SOAP και WSDL στο πλαίσιο του SOA, το οικοσύστημα εξελίχθηκε προς RESTful APIs, με έμφαση σε πόρους που εκτίθενται μέσω HTTP με χρήση μεθόδων όπως GET, POST και DELETE. Το REST καθιέρωσε αρχές όπως statelessness και uniform interfaces, επιτρέποντας σε APIs να επεκταθούν και να λειτουργήσουν αποδοτικά σε περιβάλλοντα με πολλούς πελάτες.

Η μετάβαση στο REST συνοδεύτηκε από υιοθέτηση JSON ως μορφή ανταλλαγής δεδομένων, καθιστώντας την επικοινωνία πιο ελαφριά και αναγνώσιμη. Τα APIs πλέον διαδραματίζουν ρόλο σε microservices, serverless πλατφόρμες, mobile backends, IoT συσκευές και αυτόνομα συστήματα. Ο ρόλος τους δεν είναι απλώς τεχνικός, αλλά και στρατηγικός: επιτρέπουν την ψηφιοποίηση λειτουργιών, την πρόσβαση σε τρίτους και την οικοδόμηση modular αρχιτεκτονικών.

Ο σχεδιασμός API μετατράπηκε έτσι σε αρχιτεκτονική διαδικασία. Οι δύο βασικές φιλοσοφίες που επικράτησαν είναι η Code-First και η Model-First προσέγγιση. Στην Code-First μέθοδο, οι προγραμματιστές ξεκινούν απευθείας με την υλοποίηση endpoints, χρησιμοποιώντας frameworks όπως Spring Boot ή Express.js. Η τεκμηρίωση παράγεται εκ των υστέρων μέσω εργαλείων που εξάγουν μεταδεδομένα από annotations ή decorators. Αν και η μέθοδος αυτή είναι ταχεία και οικεία στους developers, οδηγεί συχνά σε αποκλίσεις μεταξύ τεκμηρίωσης και υλοποίησης, δυσκολίες επικύρωσης και ελλιπή ορατότητα από μη τεχνικά μέλη της ομάδας.

Αντίθετα, η Model-First φιλοσοφία ξεκινά από τον ορισμό ενός επίσημου μοντέλου του API, χρησιμοποιώντας γλώσσες όπως OpenAPI, Protocol Buffers, Smithy ή TypeSpec. Το μοντέλο περιγράφει πλήρως τα endpoints, τα schemas, τα παραδείγματα και τους κανόνες, και αποτελεί κοινό σημείο αναφοράς για όλους τους εμπλεκόμενους. Μέσα από αυτό παράγονται αυτόματα τα stubs, τα SDKs και η τεκμηρίωση, εξασφαλίζοντας συνέπεια και ομοιομορφία.

Η προσέγγιση αυτή ευθυγραμμίζεται με τις αρχές του Model-Driven Engineering και προωθεί τη συνεργασία, τη συμμόρφωση με πρότυπα και την επεκτασιμότητα. Εισάγει, ωστόσο, και νέες απαιτήσεις: οι ομάδες πρέπει να μάθουν τα modeling εργαλεία, να επενδύσουν σε pipelines επικύρωσης και να διαχειριστούν repositories μοντέλων με πειθαρχία.

Ένα από τα πιο διαδεδομένα formats περιγραφής APIs είναι το OpenAPI (πρώην Swagger), το οποίο περιγράφει RESTful APIs σε YAML ή JSON. Μέσω δομών όπως paths και components, οι προγραμματιστές μπορούν να ορίσουν endpoints, schemas και authentication μηχανισμούς. Το OpenAPI υποστηρίζεται από εργαλεία όπως Swagger UI για διαδραστική τεκμηρίωση, OpenAPI Generator για παραγωγή SDKs και Stoplight για validation.

Παρά την ευρεία αποδοχή του, το OpenAPI περιορίζεται στον REST paradigm και δεν καλύπτει εγγενώς streaming ή bidirectional επικοινωνία. Η Model-First προσέγγιση επεκτείνεται πέρα από το OpenAPI, υιοθετώντας πιο εκφραστικά μοντέλα.

Συνολικά, οι δύο προσεγγίσεις εκφράζουν διαφορετικές φιλοσοφίες: η Code-First δίνει έμφαση στην ταχύτητα και την άμεση παραγωγή λειτουργικού κώδικα, αλλά θυσιάζει τη συνοχή και τη δομική αρτιότητα. Αντίθετα, η Model-First προσφέρει ένα ελεγχόμενο και τυποποιημένο περιβάλλον σχεδιασμού, ενσωματώνοντας αυτοματοποίηση και επικύρωση. Η υιοθέτηση της απαιτεί αλλαγή νοοτροπίας και πρόσθετη προσπάθεια στην αρχή, όμως επιτρέπει τη διατήρηση συμβατότητας, την κλιμάκωση και την υιοθέτηση κοινών standards σε ολόκληρο τον οργανισμό.

3. Model-First Σχεδιασμός API

Το Model-First API Design βασίζεται στην αρχή ότι η μοντελοποίηση του συμβολαίου υπηρεσίας και των δομών δεδομένων προηγείται της υλοποίησης. Η διαδικασία ξεκινά από ένα αφηρημένο, τεχνολογικά ανεξάρτητο μοντέλο που περιγράφει τις λειτουργίες, τα schemas και τους περιορισμούς. Από το μοντέλο αυτό παράγονται αυτόματα server stubs, SDKs, τεκμηρίωση και εργαλεία επικύρωσης, καθιερώνοντάς το ως τη μοναδική πηγή αλήθειας σε όλη την ανάπτυξη.

Το μοντέλο επιτρέπει πρώιμη συμμετοχή διαφόρων ρόλων στον σχεδιασμό (όχι μόνο μηχανικών), ενώ η επικύρωση "shift-left" βοηθά στον έγκαιρο εντοπισμό σφαλμάτων. Η αυτοματοποίηση επαναλαμβανόμενων εργασιών και η ομοιομορφία σε ετερογενή περιβάλλοντα καθιστούν την προσέγγιση ιδιαίτερα αποδοτική.

Το gRPC αξιοποιεί το Protocol Buffers (.proto) ως κεντρικό μοντέλο σχεδιασμού. Στα .proto αρχεία δηλώνονται τύποι, λειτουργίες RPC και υπηρεσίες, που στη συνέχεια μεταγλωττίζονται σε κώδικα για πολλές γλώσσες. Η σύζευξη του μοντέλου με το runtime του gRPC επιτρέπει αποδοτική επικοινωνία πάνω από HTTP/2, υποστηρίζοντας διαδραστικά πρότυπα (unary, streaming κ.ά.) ιδανικά για real-time και low-latency σενάρια.

Τα Protocol Buffers είναι σχεδιασμένα για compactness και συμβατότητα μεταξύ εκδόσεων. Οι αριθμητικές ετικέτες αντί για ονόματα πεδίων επιτρέπουν μικρότερο μέγεθος και γρήγορη επεξεργασία. Το runtime του gRPC προσφέρει primitives για authentication, deadlines, health checks και interceptors, υποστηρίζοντας παραγωγικές εγκαταστάσεις σε περιβάλλοντα με observability και service mesh.

Το gRPC συνδυάζεται καλά με σύγχρονες υποδομές cloud: reverse proxies, gateways και εργαλεία όπως grpc-gateway επιτρέπουν την έκθεση JSON/REST διεπαφών εξωτερικά, ενώ το εσωτερικό συμβόλαιο παραμένει σε gRPC. Η αρχιτεκτονική αυτή επιτρέπει υψηλές επιδόσεις εσωτερικά και συμβατότητα με REST εξωτερικά. Η υποστήριξη tooling είναι ευρεία και περιλαμβάνει πολλές γλώσσες, plugins, IDE integrations και εργαλεία για testing και τεκμηρίωση. Τα .proto χρησιμοποιούνται ως συμβόλαια για παράλληλη ανάπτυξη, επικύρωση αλλαγών και αυτόματη παραγωγή SDKs. Ωστόσο, υπάρχουν περιορισμοί: η δυαδική μορφή καθιστά δύσκολη τη δοκιμή χωρίς ειδικά εργαλεία· η έλλειψη πλήρους υποστήριξης από browsers απαιτεί χρήση gRPC-Web ή transcoding· και για δημόσια APIs, το REST παραμένει πιο διαδεδομένο. Συνήθεις

χρήσεις περιλαμβάνουν microservices, backend APIs και mobile backends, ενώ συχνά επιλέγεται υβριδικό μοντέλο (gRPC εσωτερικά, REST εξωτερικά).

Το Smithy της AWS είναι μια γλώσσα μοντελοποίησης APIs που διαχωρίζει σαφώς τη δομή δεδομένων (shapes) από τις σημασιολογικές ιδιότητες (traits). Το μοντέλο παραμένει ανεξάρτητο από το πρωτόκολλο, επιτρέποντας προβολές (projections) σε διαφορετικές υλοποιήσεις όπως restJson1 ή custom binary protocols. Αυτό επιτυγχάνεται μέσω traits που ορίζουν π.χ. bindings σε HTTP verbs, authentication, pagination ή error handling. Η υλοποίηση περιλαμβάνει tooling για επικύρωση, παραγωγή κώδικα και τεκμηρίωση. Μέσω του smithy-build.json δηλώνονται τα plugins που θα ενεργοποιήσουν προβολές και SDKs. Τα traits λειτουργούν ως φορείς πολιτικής: για παράδειγμα, ορίζουν status codes, συμπεριφορές pagination ή περιορισμούς πεδίων. Το Smithy χρησιμοποιείται εκτενώς από την AWS για να περιγράψει APIs και να παράγει SDKs και CLI clients, και το οικοσύστημα του επεκτείνεται σταδιακά και εκτός AWS.

Το TypeSpec της Microsoft είναι μια πιο προγραμματιστική προσέγγιση στη μοντελοποίηση APIs, με σύνταξη εμπνευσμένη από TypeScript. Συνδυάζει δηλωτικούς ορισμούς με templates και decorators, επιτρέποντας αναπαραγωγή OpenAPI, Protobuf, SDKs ή documentation από μία και μόνο πηγή. Το βασικό μοντέλο περιγράφει τους τύπους και τις λειτουργίες υπηρεσίας μέσω interfaces. Οι decorators, όπως @route, @get, @post, αποδίδουν σημασιολογικές πληροφορίες που αξιοποιούνται από emitters. Οι emitters είναι βιβλιοθήκες που μετατρέπουν το μοντέλο σε τελικά artifacts (OpenAPI specs, gRPC definitions κ.λπ.). Η υποδομή emitters είναι επεκτάσιμη, επιτρέποντας συγγραφή custom εξαγωγών χωρίς επανάληψη της λογικής μορφοποίησης.

4. Η Σημασία του Model-First Σχεδιασμός στον Κύκλο Ζωής του API

Ο παραδοσιακός κύκλος ζωής ενός API περιλαμβάνει τα στάδια του σχεδιασμού, της υλοποίησης, των δοκιμών και της παραγωγής. Συχνά όμως, αυτά τα στάδια εκτελούνται απομονωμένα, δημιουργώντας ασυνέπειες, καθυστερήσεις και τεχνικό χρέος. Η προσέγγιση Model-First επαναπροσδιορίζει τον κύκλο ζωής του API, εδραιώνοντας το μοντέλο ως το μοναδικό σημείο αναφοράς για όλη την ανάπτυξη.

Η διαδικασία ξεκινά με τη μοντελοποίηση των endpoints και των δομών δεδομένων με χρήση εργαλείων όπως Smithy, TypeSpec ή Protobuf, σε περιβάλλον ουδέτερο ως προς την τεχνολογία. Από το επικυρωμένο μοντέλο παράγονται αυτόματα stubs, SDKs και τεκμηρίωση, διασφαλίζοντας ότι η υλοποίηση είναι συγχρονισμένη με τον σχεδιασμό. Μέσω CI/CD pipelines, το μοντέλο υποβάλλεται σε ελέγχους συμβατότητας, επικύρωσης και versioning πριν από κάθε ανάπτυξη.

Το testing ξεκινά ήδη από το στάδιο σχεδίασης, με χρήση mock servers και schema validators που επιτρέπουν την πρόωρη ανίχνευση προβλημάτων. Έτσι αποφεύγονται λάθη που εντοπίζονται αργότερα στην παραγωγή. Επίσης, ο σχεδιασμός διευκολύνει την εξελικτική αναβάθμιση APIs, επιτρέποντας μη-καταστροφικές αλλαγές, απόσυρση endpoints και διαχείριση εκδόσεων, χωρίς να επηρεάζονται υπάρχοντες καταναλωτές.

Ο κύκλος ζωής διαμορφώνεται ως ένας επαναλαμβανόμενος βρόχος: ορισμός μοντέλου, επανεξέταση και επικύρωση, αυτόματη παραγωγή artifacts, υλοποίηση, ανάπτυξη και εξέλιξη. Σε αντίθεση με τη γραμμική ροή των code-first μοντέλων, η Model-First προσέγγιση συνδέει συνεχώς τον σχεδιασμό με την παραγωγή, ενισχύοντας τη διαφάνεια, τη συνέπεια και την ευθυγράμμιση τεχνικών και επιχειρησιακών στόχων.

Ένα από τα σημαντικότερα πλεονεκτήματα του Model-First είναι η ενεργή συμμετοχή μη μηχανικών ρόλων στον σχεδιασμό. Μέσα από τις μοντελοποιήσεις σε Smithy, TypeSpec ή Protobuf, ομάδες προϊόντος, σχεδιαστές UX ή μέλη κανονιστικής συμμόρφωσης μπορούν να διαβάσουν, σχολιάσουν και συνδιαμορφώσουν το API. Οι περιγραφές είναι δηλωτικές, ευανάγνωστες και εκτελούνται από εργαλεία που τις μετατρέπουν σε mock servers, τεκμηρίωση ή validators.

Αυτή η προσβασιμότητα επιτρέπει στους product managers να εξασφαλίζουν την ευθυγράμμιση με τα επιχειρησιακά σενάρια, στους σχεδιαστές να επικυρώνουν τις ροές δεδομένων και στους νομικούς να ελέγχουν τη διαχείριση προσωπικών πληροφοριών

Η εφαρμογή κοινών σχεδιαστικών μοτίβων και η κεντρική διαχείριση μοντέλων διασφαλίζουν συνέπεια και ποιότητα σε όλο το API οικοσύστημα. Επαναχρησιμοποιούμενα traits ή decorators για authentication, error handling και pagination μπορούν να εφαρμοστούν οριζόντια σε όλα τα services. Έτσι, μειώνεται ο κίνδυνος ασυνέπειας και οι ομάδες αποφεύγουν την επανεφεύρεση υφιστάμενων λύσεων.

Η διακυβέρνηση σε Model-First περιβάλλον δεν είναι στατική. Καθώς νέες τεχνολογίες όπως GraphQL federation ή AsyncAPI ενσωματώνονται, τα μοντέλα και οι validators εξελίσσονται ανάλογα. Η αρχιτεκτονική παραμένει διαφανής και ευέλικτη, προσαρμοζόμενη στις στρατηγικές ανάγκες.

Η προσέγγιση Model-First συνδέεται άμεσα με cloud-native αρχιτεκτονικές. Τα APIs αντιμετωπίζονται ως μονάδες ενοποίησης σε περιβάλλοντα multi-cloud, όπου η ανεξαρτησία από τη στοίβα υλοποίησης είναι κρίσιμη. Τα μοντέλα των APIs περιγράφονται δηλωτικά και ενορχηστρώνονται όπως η υποδομή: version-controlled, παραμετροποιημένα, και συγχρονισμένα με Terraform ή Kubernetes manifests. Οι αλλαγές στο μοντέλο επιφέρουν αυτόματες τροποποιήσεις στην υποδομή και τις πολιτικές ασφαλείας, μειώνοντας το drift και ενισχύοντας την αξιοπιστία. Πλατφόρμες όπως AWS API Gateway, Azure API Management και Google Cloud Endpoints υποστηρίζουν εισαγωγή OpenAPI ή gRPC specs, επιτρέποντας στις αλλαγές στο μοντέλο να αντικατοπτρίζονται άμεσα στο runtime.

Η διαλειτουργικότητα εξασφαλίζεται μέσω εξαγωγών σε πολλαπλά πρωτόκολλα: από ένα ενιαίο μοντέλο μπορούν να προκύψουν REST, gRPC ή GraphQL interfaces, επιτρέποντας λειτουργία σε ετερογενείς υποδομές χωρίς επανασχεδιασμό. Η φορητότητα αυτή προστατεύει από vendor lock-in και υποστηρίζει υβριδικά σενάρια microservices.

Η τεκμηρίωση, επίσης, αναβαθμίζεται σε βασικό συστατικό του σχεδιασμού. Σε αντίθεση με τις code-first προσεγγίσεις, όπου η τεκμηρίωση είναι μεταγενέστερη και συχνά ασυνεπής, στο Model-First προκύπτει άμεσα από το μοντέλο. Περιγραφές, παραδείγματα και σχόλια

προστίθενται στα endpoints, στα schemas και στους τύπους δεδομένων, και ενσωματώνονται αυτόματα στα OpenAPI ή SDKs. Η τεκμηρίωση εξελίσσεται μαζί με το μοντέλο, διατηρείται επικαιροποιημένη και υποστηρίζει auditing, ιστορικότητα και αποτύπωση του σκεπτικού πίσω από τις σχεδιαστικές αποφάσεις. Ενισχύει τη διαφάνεια, βελτιώνει την εμπειρία των χρηστών του API και μειώνει τον χρόνο ένταξης νέων ομάδων ή συνεργατών.

Συνολικά, η Model-First σχεδίαση APIs μετατρέπει τα interfaces σε στρατηγικά εργαλεία, ευθυγραμμισμένα με τις αρχές της βιώσιμης αρχιτεκτονικής, της αυτοματοποίησης και της συνεργασίας. Συνενώνει μοντελοποίηση, τεκμηρίωση και διακυβέρνηση σε ένα ενιαίο οικοσύστημα, ενδυναμώνοντας την εξέλιξη APIs σε μεγάλη κλίμακα και με σταθερή ποιότητα.

5. Ανάλυση Εξαρτήσεων με Χρήση Model-First API Προσεγγίσεων

Οι παραδοσιακές προδιαγραφές API όπως το OpenAPI ή τα Postman Collections περιγράφουν πλήρως μεμονωμένα endpoints, αλλά σπάνια αποτυπώνουν πώς τα δεδομένα ρέουν μεταξύ τους, δηλαδή πώς η έξοδος ενός endpoint γίνεται είσοδος σε άλλο. Η κατανόηση αυτών των εξαρτήσεων μεταξύ endpoints είναι κρίσιμη για τον σχεδιασμό σύνθετων ροών, τη δοκιμή και την εξέλιξη μεγάλων APIs.

Η εργασία επεκτείνει την έννοια της στατικής ανάλυσης εξαρτήσεων στα πλαίσια Model-First σχεδιασμού API**:** Smithy**,** TypeSpec και gRPC / Protocol Buffers, αξιοποιώντας την εγγενή τυποποίηση και τη σημασιολογική δομή αυτών των μοντέλων για ακριβή και δομημένη εξαγωγή εξαρτήσεων. Η βασική λογική είναι η εξής: αν δύο endpoints μοιράζονται τύπους δεδομένων (π.χ. ένα πεδίο της απόκρισης του πρώτου χρησιμοποιείται στην είσοδο του δεύτερου), τότε υπάρχει εξάρτηση μεταξύ τους.

Το Smithy προσφέρει μία ισχυρή, τυποποιημένη γλώσσα περιγραφής REST APIs, στην οποία κάθε λειτουργία συνοδεύεται από traits (όπως `smithy.api#http`) που δηλώνουν HTTP μεθόδους, routes και παραμέτρους. Το μοντέλο περιγράφει πλήρως το API χωρίς να απαιτείται κώδικας, και οι τύποι εισόδου/εξόδου ορίζονται μέσω δομών. Η ανάλυση εξαρτήσεων αξιοποιεί το παραγόμενο Abstract Syntax Tree (AST) σε μορφή JSON. Το AST διατηρεί όλα τα περιεχόμενα του μοντέλου (τύπους, traits, δομές, σχέσεις) και επιτρέπει τη συστηματική αναγνώριση των shared shapes μεταξύ εξόδου ενός endpoint και εισόδου κάποιου άλλου. Το εργαλείο ανάλυσης δημιουργεί έναν κατευθυνόμενο γράφο εξαρτήσεων**,** όπου κόμβοι είναι τα operations και ακμές οι εξαρτήσεις μέσω κοινών τύπων ή παραμέτρων.

Το TypeSpec περιγράφει REST APIs με χρήση decorators (`@get`, `@post`, `@route` κ.ά.) για να αποτυπώσει τις διαδρομές, τις παραμέτρους και τις μεθόδους των endpoints. Οι τύποι ορίζονται δηλωτικά, με δυνατότητα scalar wrapping, δηλαδή οι primitive τύποι (όπως `string`) επεκτείνονται σε ονομαστικές μορφές (`scalar Name extends string`) για να αποδώσουν σημασιολογική πληροφορία. Παρότι δεν εκθέτει AST, το TypeSpec εξάγει OpenAPI documents που διατηρούν τις αναφορές στους εκτεταμένους τύπους. Η ανάλυση εξαρτήσεων εφαρμόζεται πάνω σε αυτά τα OpenAPI αρχεία, εντοπίζοντας κοινά `$ref` schemas ανάμεσα σε response και

request πεδία. Έτσι, οι διασυνδέσεις μεταξύ endpoints εντοπίζονται μέσω κοινών component schemas**,** όπως ακριβώς και στο Smithy.

Τα Protocol Buffers (Protobuf) βασίζονται σε RPC-style APIs και όχι σε REST, μοντελοποιώντας services και μεθόδους (RPCs) με αυστηρά ορισμένους τύπους `request` και `response`. Με την επέκταση `google.api.http` μπορούν να χαρτογραφηθούν σε HTTP endpoints, επιτρέποντας εξαγωγή σε OpenAPI μέσω του protoc compiler.

Η ανάλυση εξαρτήσεων εφαρμόζεται σε δύο λειτουργίες:

- Component Mode**:** Εντοπίζονται κοινά message types μεταξύ των αποκρίσεων και εισόδων RPCs.
- Primitive Mode**:** Επιπλέον σύγκριση γίνεται σε επίπεδο primitive πεδίων (π.χ. string `order_id`) ώστε να ανιχνευθούν περισσότερες πιθανές εξαρτήσεις.

Σε αντίθεση με το Smithy και το TypeSpec, το Protobuf δεν υποστηρίζει wrapping των primitive τύπων. Έτσι, οι τύποι είναι λιγότερο σημασιολογικά διακριτοί, γεγονός που καθιστά την ανάλυση εξαρτήσεων λιγότερο ακριβή. Παρ' όλα αυτά, με τη χρήση OpenAPI εξαγωγής και κατάλληλης ανάλυσης, εντοπίζονται ουσιαστικές σχέσεις μεταξύ RPCs.

6. Case Study: Ανάλυση Εξαρτήσεων και Σύγκριση στο PayPal API

Για την αξιολόγηση της προτεινόμενης μεθοδολογίας ανίχνευσης εξαρτήσεων μεταξύ endpoints, επιλέχθηκε ως μελέτη περίπτωσης το PayPal REST API. Η ροή εργασίας περιλαμβάνει δημιουργία και ενημέρωση προϊόντος, υποβολή παραγγελίας, διαχείριση αποστολής και πληρωμής, και προσθήκη στοιχείων παρακολούθησης. Σε κάθε στάδιο, τα δεδομένα που παράγονται (όπως `Product_Id`, `Order_Id`, `Payment_Id`, `Tracking_Id`) χρησιμοποιούνται σε επόμενα βήματα, σχηματίζοντας μια αλυσίδα εξαρτήσεων που αντικατοπτρίζει τον πραγματικό κύκλο ζωής μιας παραγγελίας. Το ίδιο σύνολο endpoints μοντελοποιήθηκε με τρεις διαφορετικές Model-First τεχνολογίες: Smithy, TypeSpec και gRPC/Protobuf. Η υλοποίηση διατήρησε κοινά ονόματα, paths και δομές δεδομένων, ώστε η σύγκριση να είναι αντικειμενική. Στα επιλεγμένα endpoints περιλαμβάνονται λειτουργίες όπως δημιουργία παραγγελίας (POST /v2/checkout/orders), έγκριση πληρωμής (POST /v2/checkout/orders/{order_id}/authorize), ανάκτηση στοιχείων πληρωμής (GET /v2/payments/captures/{capture_id}), και προσθήκη tracking (POST /v1/shipping/trackers-batch).

Η εξαγωγή εξαρτήσεων πραγματοποιήθηκε με τους analyzers του προηγούμενου κεφαλαίου, προσαρμοσμένους για κάθε μοντέλο. Η ανάλυση συγκρίνει τους τύπους εξόδου κάθε endpoint με τους τύπους εισόδου των υπολοίπων. Αν βρεθεί κοινός τύπος ή schema, καταγράφεται κατευθυνόμενη εξάρτηση από τον παραγωγό (source) προς τον καταναλωτή (target).

Στο Smithy, το εργαλείο ανάλυσης αξιοποίησε το Abstract Syntax Tree που παράγεται κατά την μεταγλώττιση, με τους τύπους να διατηρούν πλήρη σημασιολογική ταυτότητα (π.χ. `smithy.paypal#Order_Id`). Αυτό επέτρεψε την αποτύπωση των εξαρτήσεων με ακρίβεια, χωρίς ασάφειες λόγω κοινού primitive τύπου. Το αποτέλεσμα ήταν ένα πλήρες γράφημα εξαρτήσεων που βασίζεται σε ισχυρές δομές τύπων και όχι σε σύγκριση ονομάτων.

Το TypeSpec λειτούργησε με παρόμοιο τρόπο. Οι scalar τύποι όπως `scalar Order_Id extends string;` διατηρήθηκαν στο OpenAPI ως `$ref` schemas, διατηρώντας τη σημασιολογική πληροφορία. Η ανάλυση εντόπισε με ακρίβεια όλες τις εξαρτήσεις, τόσο για απλούς τύπους όσο και για σύνθετες οντότητες όπως `Amount`, `Payment`, `Tracker`. Τα αποτελέσματα ταυτίστηκαν πλήρως με εκείνα του Smithy.

Στο gRPC/Protobuf, η εξαγωγή OpenAPI έγινε μέσω protoc, με τα .proto αρχεία να μετατρέπονται σε JSON schemas. Ωστόσο, το Protobuf δεν υποστηρίζει semantic wrapping σε primitive πεδία. Αυτό σημαίνει πως πεδία όπως `order_id` αναπαρίστανται ως απλά strings, χωρίς δυνατότητα διάκρισης από άλλα παρόμοια πεδία. Η ανάλυση εξαρτήσεων στηρίχθηκε είτε μόνο σε structured message types (component mode) είτε και σε primitive πεδία (primitive mode). Στην πρώτη περίπτωση, τα αποτελέσματα ήταν ικανοποιητικά, με καθαρές σχέσεις μεταξύ RPCs. Στη δεύτερη, η ανάλυση παρήγαγε αρκετό "θόρυβο", με ψευδείς θετικές εξαρτήσεις λόγω κοινών ονομάτων, αλλά διαφορετικής σημασίας.

Τα Smithy και TypeSpec απέδωσαν πρακτικά ταυτόσημα γραφήματα εξαρτήσεων (Figures 4 και 5), με σαφήνεια και χωρίς υπερβολικές ή λανθασμένες συσχετίσεις. Το gRPC εμφάνισε χαμηλότερη σημασιολογική συνοχή, ιδίως όταν περιλαμβάνονταν primitives στην ανάλυση (Figure 7), ενώ σε component-only mode τα αποτελέσματα ήταν πιο αξιόπιστα (Figure 6).

Η σύγκριση με τις στατικές και δυναμικές αναλύσεις του εργαλείου RADAR ανέδειξε την ανωτερότητα της Model-First προσέγγισης. Η στατική ανάλυση του RADAR βασίζεται σε συνωνυμίες ονομάτων και παραλείπει σημαντικές εξαρτήσεις (Figure 8). Η δυναμική ανάλυση καταγράφει μόνο ό,τι παρατηρείται σε πραγματικό χρόνο κατά την εκτέλεση ενός use case (Figure 9), οπότε περιορίζεται από το ποια σενάρια ενεργοποιούνται. Για παράδειγμα, στο endpoint Create Order, το Smithy και η δυναμική ανάλυση εντόπισαν όλες τις εξαρτήσεις (Figures 10 και 12), ενώ η στατική προσέγγιση εντόπισε μόνο μία (Figure 11). Αντίστροφα, στο Create Product, η Smithy και η στατική ανάλυση κατέγραψαν μία κοινή εξάρτηση, ενώ η δυναμική δεν την εντόπισε καθόλου λόγω απουσίας εκτέλεσης της ροής (Figures 13–15).

Συνολικά, η Model-First προσέγγιση προσφέρει τον συνδυασμό σημασιολογικής ακρίβειας και πληρότητας χωρίς να απαιτεί runtime δεδομένα. Παρέχει αξιόπιστα γραφήματα εξαρτήσεων σε επίπεδο αντικειμένων και εννοιών, ενώ οι στατικές και δυναμικές μέθοδοι του RADAR είτε χάνουν πληροφορία είτε περιορίζονται από την εκτέλεση.

7. Συμπεράσματα

Η εργασία ανέδειξε το μοντέλο Model-First ως μια σύγχρονη και αξιόπιστη προσέγγιση στον σχεδιασμό APIs, προτάσσοντας την ύπαρξη ενός ενιαίου μοντέλου ως βάση για κώδικα και documentation. Αναλύθηκαν τα gRPC, Smithy και TypeSpec, με το gRPC να υπερέχει σε εσωτερικές, low-latency επικοινωνίες, αλλά να υστερεί στην εννοιολογική αναπαράσταση. Smithy και TypeSpec προσφέρουν μεγαλύτερη εκφραστικότητα και ευελιξία, με το Smithy να έχει πιο ώριμο οικοσύστημα και το TypeSpec πιο σύγχρονη εμπειρία χρήστη. Μέσω της στατικής ανάλυσης εξαρτήσεων, αποδείχθηκε ότι τα μοντέλα αυτά μπορούν να αποδώσουν ακριβή

γραφήματα σχέσεων μεταξύ endpoints χωρίς ανάγκη για runtime δεδομένα. Τέλος, προτείνεται ως μελλοντική κατεύθυνση η χρήση των Smithy/TypeSpec για τον σχεδιασμό MCP servers, αξιοποιώντας τη δυναμική του model-first και στον χώρο της αλληλεπίδρασης AI εργαλείων.

1 – Introduction

1.1 Background and Motivation

In today's interconnected digital ecosystem, Application Programming Interfaces (APIs) have become the fundamental building blocks that enable software systems to communicate, share data, and deliver integrated services. They are no longer auxiliary technical artifacts but core strategic assets that define how organizations expose functionality, build ecosystems, and foster innovation [1].

The accelerated growth of distributed architectures, microservices, serverless computing, and cloud-native applications, has increased both the scale and complexity of API development. In such environments, maintaining consistency, interoperability, and clarity across hundreds of interdependent endpoints becomes a key engineering and governance challenge [2]. APIs now serve not only as technical interfaces but also as contracts between teams, influencing business models, security boundaries, and lifecycle automation.

Historically, most APIs were created following a code-first paradigm, in which engineers implemented functionality in code and later exposed the interface through documentation or specification tools such as Swagger or OpenAPI. While effective for small systems, this approach often produces fragmented ecosystems: documentation drifts from implementation, design feedback occurs late, and collaboration among cross-functional teams is limited.

In response, modern organizations have embraced the Model-First (or Design-First) paradigm [3], [4], [5]. In this approach, the API is defined formally and semantically before implementation, using modeling languages such as Protocol Buffers (gRPC), Smithy, and TypeSpec. The model becomes the single source of truth from which code, documentation, SDKs, and tests are automatically generated. This practice ensures early validation, architectural consistency, and seamless collaboration between technical and non-technical stakeholders.

Furthermore, as APIs increasingly operate across multi-cloud and hybrid infrastructures, a consistent model-driven foundation enables cloud-agnostic deployment and automated lifecycle governance [6]. The convergence of Model-First design with DevOps automation and cloud management represents a paradigm shift: APIs are now treated as versioned, governed, and continuously validated artifacts much like infrastructure-as-code.

This thesis explores that paradigm in depth. It studies the leading Model-First frameworks, gRPC/Protocol Buffers, Smithy, and TypeSpec, and proposes a methodology that leverages these models for accurate static endpoint-interdependency analysis. The research combines theoretical analysis, framework comparison, and empirical validation through a real-world PayPal API case study, extending the foundational RADAR work on RESTful dependency detection [7], [8], [9].

1.2 Objectives of the Thesis

The main objective of this thesis is twofold:

1. To perform a comparative analysis of the three dominant Model-First API design frameworks, gRPC, Smithy, and TypeSpec, evaluating their architecture, expressiveness, interoperability, and integration with the cloud ecosystem.
2. To develop and assess a methodology for static inter-endpoint dependency analysis based on Model-First representations, improving upon the static and dynamic methods previously proposed in the RADAR research framework [11].

More specifically, this work aims to:

1. Demonstrate how Model-First artifacts (e.g., Smithy ASTs, TypeSpec schemas, gRPC OpenAPI exports) can be analyzed to discover inter-endpoint dependencies without requiring runtime traces.
2. Compare the resulting dependency graphs against RADAR's static and dynamic analyses, validating improvements in accuracy and semantic cohesion.
3. Highlight how the Model-First approach provides noise-free, type-aware, and higher-level dependency detection by leveraging formal schema structures rather than textual name matching.

Ultimately, the thesis bridges the gap between API design theory and analytical tooling, showing how Model-First frameworks can be repurposed not only for specification and generation but also for automated reasoning and knowledge extraction across large-scale API ecosystems.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows:

- **Chapter 1 – Introduction** outlines the motivation, objectives, and scope of the thesis, introducing Model-First API Design and the selected frameworks.
- **Chapter 2 – Fundamentals of API Design** introduces the foundational concepts of APIs and Web Services, explaining the evolution from code-first to model-first design and discussing classical specification methods such as OpenAPI and Swagger.
- **Chapter 3 – Model-First API Design Paradigms** provides an in-depth analysis of the three principal frameworks: gRPC/Protocol Buffers, Smithy, and TypeSpec, focusing on their modeling syntax, architectural abstractions, and design philosophies.
- **Chapter 4 – The Importance of Model-Design-First in the API Lifecycle and Collaboration** explores how Model-First design transforms collaboration, governance, consistency enforcement, and documentation generation across the API lifecycle.
- **Chapter 5 – Static Analysis of Inter-Endpoint Dependencies** introduces the methodology for extracting endpoint dependencies from Model-First artifacts and presents the analytical foundations for Smithy, TypeSpec, and gRPC models.
- **Chapter 6 – Case Study: PayPal API Dependency Analysis** applies the developed methodology to a real-world PayPal API scenario, comparing the results against

RADAR's static and dynamic analyses and discussing the accuracy and interpretability of each approach.

- **Chapter 7 – Conclusion** summarizes the findings, compares the frameworks, and highlights future directions for model-driven dependency analysis.

2 – Fundamentals of API Design

2.1 Overview of APIs and Web Services

Application Programming Interfaces (APIs) constitute the foundational layer of modern software systems, enabling structured and standardized interaction between independent software components. An API defines a formal contract that describes how a client can communicate with a service, specifying available operations, the expected inputs and outputs, and the semantics that govern their interaction [1]. This concept has evolved from simple procedural calls in monolithic systems to a sophisticated architecture paradigm that underpins today's distributed and cloud-native environments.

In the early stages of distributed computing, the Service-Oriented Architecture (SOA) model dominated enterprise integration. APIs were primarily implemented using SOAP (Simple Object Access Protocol), with message formats defined in XML and contracts specified in WSDL (Web Services Description Language). While SOAP provided strong type safety and extensibility, its reliance on verbose XML schemas and strict contracts introduced significant overhead, limiting agility and increasing development complexity. The architectural shift toward Representational State Transfer (REST), formalized by Fielding [10], marked a significant transition in API philosophy. REST introduced a resource-oriented architecture, in which each resource is identified by a URI and manipulated through standardized HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`. It emphasized statelessness, cacheability, and uniform interfaces, enabling APIs to scale horizontally and integrate seamlessly across diverse clients.

Over time, RESTful APIs became the backbone of web-based systems and microservices. Their reliance on JSON (JavaScript Object Notation) made data exchange lightweight and human-readable. Today, APIs extend far beyond web applications, they power microservices, serverless architectures, mobile backends, IoT ecosystems, and even machine-to-machine communication within autonomous systems. Enterprises use APIs to expose digital capabilities, facilitate third-party integration, and enable composable architectures that accelerate innovation. This ubiquity, however, introduces new challenges. APIs must now be designed with governance, security, discoverability, and backward compatibility in mind. Maintaining consistency across thousands of endpoints in multi-cloud environments requires structured methodologies that move beyond ad hoc development practices.

This evolution has transformed API design from a programming task into a core architectural discipline. It now demands formalized approaches that balance technical precision, scalability, and organizational agility. Two dominant philosophies have emerged in this context: the Code-First and the Model-First approaches.

2.2 API Design Approaches: Code-First vs. Model-First

Modern API design methodologies can be categorized based on whether implementation precedes specification or specification precedes implementation. The Code-First approach emphasizes quick development and bottom-up generation of documentation, while the Model-First approach focuses on defining the API as a formal model from which all implementations derive.

In the Code-First approach, developers start by writing code that implements the API endpoints. Frameworks such as Spring Boot, Flask, and Express.js enable rapid creation of HTTP routes, controllers, and serialization logic. Once the core functionality is in place, automated tools extract metadata from annotations or decorators to produce formal documentation, typically in OpenAPI (Swagger) format.

This approach excels in speed and developer familiarity, making it highly effective for prototyping or internal APIs where tight deadlines outweigh documentation rigor. However, several long-term drawbacks emerge as systems scale. Documentation often drifts from implementation, as developers make changes to code without regenerating or verifying the specification. Validation becomes reactive, with schema mismatches and missing parameters identified only during runtime testing. Moreover, since the design is embedded in the codebase, collaboration across multidisciplinary teams becomes difficult, excluding product owners, UX designers, or compliance auditors who lack access to or familiarity with the source code. As systems evolve, the lack of a unified specification leads to fragmentation, inconsistent data models, and significant maintenance overhead.

The Model-First paradigm reverses this workflow. Here, the process begins with a formal definition of the API contract, typically written in a modeling language such as OpenAPI, Protocol Buffers (gRPC), AWS Smithy, or Microsoft TypeSpec. This model serves as the authoritative artifact from which server stubs, client SDKs, tests, and documentation are automatically generated [3], [4], [5]. The model defines all aspects of the API including endpoints, parameters, data types, request and response bodies, and even example payloads. Because it is both machine- and human-readable, it becomes a shared reference point for engineers, architects, and business stakeholders alike.

Model-First design aligns with the principles of Model-Driven Engineering (MDE), where high-level abstractions guide implementation through automated transformations. By defining the structure and semantics upfront, teams can perform early validation, enforce consistency, and generate downstream artifacts automatically. This approach promotes collaboration and governance, ensuring that all services adhere to organizational standards. It also decouples design from implementation, making it easier to refactor APIs, enforce backward compatibility, and support multi-language client generation. The tradeoff lies in the higher upfront cost, since teams must invest time in defining schemas, learning modeling tools, and integrating validation pipelines but the long-term gains in consistency and scalability are substantial.

2.3 Classic Specification Methods: OpenAPI and Swagger

The OpenAPI Specification (OAS), formerly known as Swagger, is the most widely adopted standard for describing RESTful APIs. It provides a language-agnostic, machine-readable format written in YAML or JSON that defines the structure, parameters, authentication methods, and data models of an API [11]. OpenAPI facilitates clear communication between humans and machines by serving as a contract that defines how an API behaves, without requiring access to its implementation.

An OpenAPI document is structured around the concept of paths, which represent individual endpoints, and components, which define reusable schema objects. Each path can include multiple HTTP operations (`GET`, `POST`, `PUT`, `DELETE`), each of which specifies the expected inputs and outputs. For example:

```yaml
paths:
  /orders/{orderId}:
    get:
      summary: Retrieve order details
      parameters:
        - name: orderId
          in: path
          required: true
          schema:
            type: string
        - name: expand
          in: query
          required: false
          schema:
            type: string
            enum: [items, payments]
      responses:
        '200':
          description: Order retrieved successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'
components:
  schemas:
    Order:
      type: object
      required:
        - id
        - amount
      properties:
        id:
          type: string
        amount:
          type: number
        currency:
          type: string
```

```
38.
```

This specification defines a `GET /orders/{orderId}` endpoint that retrieves an order's details. It accepts a path parameter (`orderId`) identifying the resource and an optional query parameter (`expand`) that modifies the response. The response schema references a reusable object named `Order` defined under `components/schemas`. This modular design allows complex APIs to maintain consistency across hundreds of endpoints.

The OpenAPI ecosystem includes tools for interactive documentation (Swagger UI), automatic SDK generation (OpenAPI Generator) and validation (Stoplight). [12], [13], [14]. However, OpenAPI's expressiveness is limited to RESTful paradigms since it cannot natively represent streaming, bidirectional communication, or protocol-agnostic services.

2.4 Advantages and Challenges of API Design Approaches

The Code-First and Model-First methodologies represent fundamentally different strategies for managing complexity in API ecosystems. The Code-First method emphasizes immediacy and familiarity. Developers can quickly create and deploy endpoints with minimal process overhead. The implementation naturally mirrors the development workflow, allowing rapid experimentation. However, this flexibility comes at a cost: as APIs scale, code-first systems struggle with documentation divergence, inconsistent versioning, and poor visibility across organizational boundaries. The specification becomes an afterthought, and when APIs evolve independently across teams, inconsistencies in naming conventions, response structures, and authentication mechanisms accumulate. This fragmentation ultimately hampers maintainability and integration.

By contrast, the Model-First approach introduces rigor and consistency through explicit modeling. The API definition becomes the foundation of the software lifecycle: a single, verifiable artifact that ensures alignment between design, implementation, and consumption. Because the model serves as the canonical source of truth, tools can automatically generate server stubs, SDKs, mock servers, and documentation, maintaining complete synchronization across development environments. This paradigm also enables early validation, allowing teams to test contracts before implementation. It further enhances governance, as shared schemas and organizational patterns can be enforced across multiple teams through centralized design systems.

Nevertheless, the Model-First paradigm introduces new challenges. Defining APIs as models requires a shift in mindset: teams must invest time in training, adopt schema management tools, and enforce automated validation in CI/CD pipelines. The need for dedicated model repositories and strict versioning introduces operational overhead. However, these challenges represent an investment in maturity, as they enable predictability, interoperability, and controlled evolution of APIs over time.

In summary, Code-First approaches prioritize speed and simplicity, while Model-First approaches emphasize structure, automation, and collaboration. The two philosophies reflect different trade-offs between short-term agility and long-term sustainability. As the complexity of

distributed systems grows, the Model-First paradigm is increasingly favored for its ability to produce consistent, maintainable, and evolvable APIs across large organizations.

The next chapter explores the Model-First design frameworks in detail, focusing on gRPC/Protocol Buffers, AWS Smithy, and Microsoft TypeSpec.

3 – Model-First API Design Paradigms

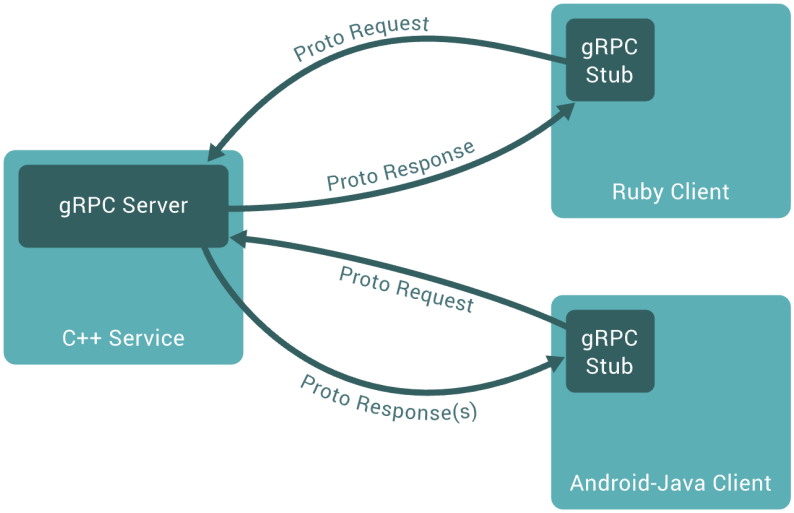
3.1 Definition and Principles of Model-First Design

Model-first API design refers to a methodology in which the definition of the service contract and its associated data structures precedes the implementation of business logic. Instead of starting from code and exposing endpoints afterward, the process begins with an abstract model that captures operations, request and response messages, and structural constraints. This model becomes the single source of truth from which server stubs, client SDKs, validation tools, and documentation are automatically derived.

The guiding principles of this paradigm include the idea of a single canonical specification, automation of repetitive engineering tasks, and consistency across heterogeneous systems. In addition, by exposing the model early in the lifecycle, collaboration is extended beyond engineers: architects, designers, and even product stakeholders can provide feedback on the contract. A further benefit lies in the so-called "shift-left validation," where potential inconsistencies or design flaws are detected at the specification level before costly implementation efforts are undertaken.

3.2 gRPC / Protocol Buffers — Architecture, Features, and Use Cases

gRPC is a modern Remote Procedure Call framework that couples a compact, schema-driven serialization format (Protocol Buffers, or *protobuf*) with the performance and semantics of HTTP/2. At design time the API surface is defined in `.proto` files: operations (RPC methods), message types, enums and services are declared there and then compiled into idiomatic client and server bindings for multiple languages. The `.proto` model therefore becomes the canonical contract for both implementation and generated artifacts (client SDKs, server skeletons, docs), which perfectly aligns with a model-design-first workflow where the specification is the single source of truth [5].



A minimal `.proto` example illustrates the central ideas where types and RPCs are declared in a compact, strongly typed syntax, and each field carries a numeric tag used by the wire format for compactness and compatibility:

```
syntax = "proto3";

package example.weather;

// Service definition: RPC methods with request/response types
service WeatherService {
  rpc GetCurrent (GetCurrentRequest) returns (GetCurrentResponse);
  rpc StreamForecast (ForecastRequest) returns (stream ForecastChunk);
}

// Message definitions: strongly-typed schemas
message GetCurrentRequest {
  string city = 1;
}

message GetCurrentResponse {
  float temperature = 1;
  string units = 2;
}

message ForecastRequest {
  string city = 1;
}

message ForecastChunk {
  string day = 1;
  float tempHigh = 2;
}
```

The `.proto` is the design artifact from which `protoc` (the Protocol Buffers compiler) generates language-specific code; `protoc` is extensible via plugins so teams can add custom generators or integrate with tools like grpc-gateway to expose a REST façade.

Architecturally, gRPC builds on HTTP/2 which provides multiplexed streams over a single TCP connection, header compression and flow control. Those transport features directly enable gRPC's support for four interaction styles: unary (request/response), server streaming, client streaming and bidirectional streaming. The richness of these interaction models makes gRPC a natural fit where long-lived channels, backpressure, or real-time streaming semantics are required (for example telemetry ingestion, real-time feeds, or bidirectional control channels).

Protocol Buffers themselves are designed for compactness and forward/backward compatibility. Because field names are not transmitted on the wire and each field is encoded using its numeric tag, protobuf payloads are typically smaller than equivalent JSON payloads and parse faster; this reduces bandwidth and CPU cost in high-volume systems. Protobuf also provides concrete, well-documented compatibility rules (e.g., safe ways to add fields, reserved ranges, and

the recommendation to favour certain tag ranges for frequently set fields because smaller tags encode into fewer bytes). These encoding and evolution rules are central to designing APIs that can evolve without breaking deployed clients.

gRPC is not merely a serialization + transport combination; it offers a mature runtime and operational primitives that production services rely on. Metadata (an HTTP-like header/trailer channel), deadlines/timeouts, cancellation propagation, interceptors (middleware), health checking and reflection are part of the gRPC ecosystem. Metadata allows authentication tokens, tracing IDs or other contextual headers to be passed alongside requests; deadlines and cancellation allow clients to bound work and prevent resource exhaustion; interceptors provide a canonical hook for cross-cutting concerns (logging, auth, metrics); and a standard health-checking protocol enables orchestrators and load-balancers to detect unhealthy backends. These primitives support robust production deployments when combined with observability and service mesh tooling [15], [16], [17].

Operationally, gRPC integrates well with modern cloud infrastructure. Service meshes (Envoy, Istio), reverse proxies and API gateways can act as ingress/egress points, perform JSON to gRPC transcoding, and provide policy, routing and observability at the network edge. Projects such as gRPC-Gateway allow teams to expose a JSON/HTTP API to external clients while keeping an internal gRPC contract for service-to-service traffic (Figure 1). This pattern lets organizations combine the performance benefits of gRPC inside the data center with the accessibility of REST for external consumers [18].

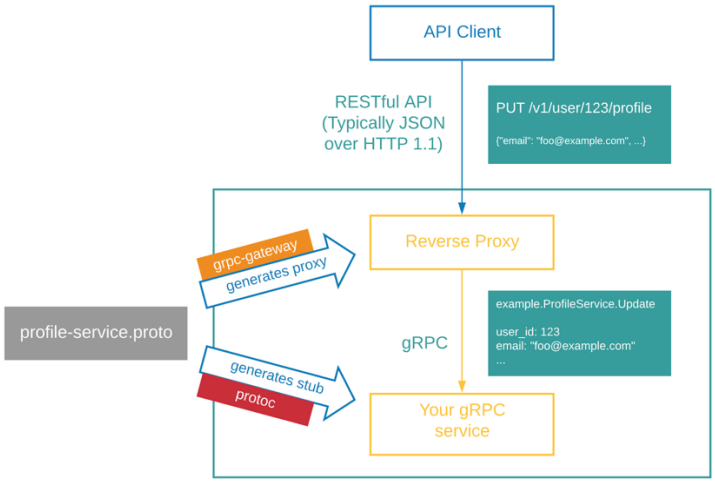


Figure 1 grpc-gateway

Tooling and language support are mature. The official Protocol Buffers toolchain supports code generation for many mainstream languages (C++, Java, Go, Python, C#, Ruby, Kotlin, Dart, PHP and more), and `protoc` is designed to be extended with plugins for additional languages or artifacts. The gRPC project supplies idiomatic libraries for numerous languages and language

communities have produced ecosystem tooling like test clients (e.g., `grpcurl`), IDE integrations, and documentation generators, making gRPC practical across polyglot environments. [19]

The model-first benefits of gRPC show up in development workflows. Because the `.proto` is the contract, teams can generate server skeletons and client libraries early and use generated mock/stub servers for parallel development. This allows front-end or integration teams to work against a mock implementation while backend services are implemented. It also enables contract validation in CI, where `protoc` can be run during pre-merge checks to enforce style or schema rules, and schema diffs can be used to block breaking changes. The compile-time nature of protobuf encourages disciplined schema evolution and integrates smoothly with automated SDK release pipelines.

However, there are important limitations and trade-offs. First, the binary wire format is not human readable, so debugging and exploratory testing require supporting tools (e.g., `grpcurl`, Postman's gRPC client) or a JSON transcoding layer. Second, native browser support for the full HTTP/2 based gRPC protocol is missing because browsers do not expose the low-level control over HTTP/2 frames required to implement the full gRPC spec, so gRPC-Web or a JSON transcoder must be used to reach browser clients [20]. Third, while gRPC and protobuf are mature within internal microservices and cloud providers, public REST + OpenAPI ecosystems still have broader tool and human-facing support (browsers, API marketplaces, third-party tools).

Typical production use cases for gRPC include internal microservice RPC, high-throughput backend APIs, low-latency request paths, real-time streams and mobile/back-end communications where compact payloads reduce bandwidth costs. Conversely, when public, browser-centric APIs or human-readable payloads are primary concerns, teams often either front gRPC services with HTTP/JSON gateways or choose REST/OpenAPI as the primary public contract. The pragmatic hybrid approach of internal gRPC contracts and public REST facades preserves both developer productivity and external accessibility [21].

In summary, gRPC + Protocol Buffers is a high-performance, model-first platform that excels when strong typing, compact binary serialization, streaming semantics and language-agnostic code generation are priorities. Its operational primitives (metadata, deadlines, interceptors, health checks), mature toolchain, and integration with cloud-native networking primitives make it a compelling choice for large-scale, internal distributed systems. Teams must, however, weigh the added complexity of binary protocols and browser limitations against the performance and contract-discipline benefits.

3.3 Smithy — Service Modeling, Traits, and Protocols

Smithy is an interface-definition and modelling language that treats the API *model* as a first-class artifact rather than an incidental byproduct of code. A Smithy model is composed of *shapes* (the typed building blocks for strings, numbers, structures, lists, maps, unions, enums, resources and services) and *traits* (metadata annotations that attach semantic meaning to shapes). This separation of shape and trait lets teams describe not only the structure of data but also validation

rules, HTTP bindings, error semantics, authentication requirements, pagination behavior and other operational concerns in a declarative way; the Smithy spec documents these core concepts and the shape/trait model in detail.

Because Smithy is protocol-agnostic, a single model can be projected onto multiple concrete protocols and bindings. Protocols and serialization rules (for example `restJson1`, `restXml`, or custom protocols) are expressed as part of the model via protocol-related traits; code generators and projection tools consume the model plus protocol traits to produce REST/JSON endpoints, binary RPC bindings, SDKs or documentation. This design establishes a clean Platform-Independent Model (PIM) → Platform-Specific Model (PSM) transformation pipeline: the platform-independent Smithy model serves as the source, validators and build tooling enforce constraints, and build plugins generate protocol-specific artifacts. The Smithy specification and official tooling guides describe how protocol and serialization traits define these projections.

A minimal Smithy IDL example demonstrates how shapes, operations and HTTP bindings are represented in practice:

```
$version: "2"
namespace example.weather

service WeatherService {
  version: "2025-06-01"
  operations: [GetCurrent]
}

@http(method: "GET", uri: "/weather/{city}")
operation GetCurrent {
  input: GetCurrentInput
  output: GetCurrentOutput
}

structure GetCurrentInput {
  @httpLabel
  city: String
}

structure GetCurrentOutput {
  temperature: Float
  units: String
}
```

This snippet shows the explicit binding of an operation to an HTTP verb and URI via the `@http` trait and how `@httpLabel` marks a member used in the path. The Smithy IDL and trait reference detail many such bindings (e.g., streaming, headers, query bindings, authentication traits) [3].

Tooling around Smithy is focused on model validation, projection, and code generation. The Smithy CLI and `smithy-build` use a `smithy-build.json` configuration to declare what model projections (called *projections* or *plugins*) should be produced; code generators are implemented as smithy-build plugins that can create language-idiomatic SDKs, server stubs and docs.

One of Smithy's operational strengths is its expressive trait ecosystem. Smithy defines many first-class traits (documentation traits, constraint traits, behavior traits, resource traits, authentication traits and protocol/serialization traits) that let model authors encode cross-cutting policies into the model itself. These traits feed downstream validators and generators: for example, pagination can be modelled with the `@paginated`/`outputToken` style, error shapes can be annotated with HTTP status mappings, and custom organization-level traits can be added to enforce naming or security rules.

Smithy is also widely used at AWS: the provider uses Smithy to model many of its public service APIs and to generate the official AWS SDKs and CLI artifacts. Recently AWS published and expanded its public set of Smithy API models, underscoring that Smithy is not only an internal modeling tool but also part of the delivery pipeline for real-world, large-scale cloud APIs. That real-world adoption translates into practical benefits: consistently generated SDKs across languages, centralized policy enforcement, and a single source of truth for documentation and code generation [22].

Despite these strengths, Smithy has limitations and practical trade-offs. Its ecosystem and most mature toolchains are closely tied to AWS; while the project is open source and used outside Amazon, external adoption is not as widespread as OpenAPI. Generators for some languages or niche use cases may be community-driven or still maturing, and teams need to invest in smithy-build configuration, custom plugins or internal generators to realize the full benefits. Additionally, the power and flexibility of traits introduce governance complexity: teams must discipline trait usage and maintain plugin implementations to ensure consistent behavior of custom semantics. Finally, while Smithy's abstractions reduce protocol lock-in, the choice of protocol projection still requires careful design to avoid semantic mismatches between model intent and concrete bindings [23].

3.4 TypeSpec — API Modeling and Language-Specific SDK Generation

TypeSpec (formerly *Cadl*) is a modern, author-centric API modeling language that treats API design as a first-class, programmatic artifact. Its syntax is intentionally familiar to TypeScript users and mixes declarative model definitions with composable programmatic constructs (models, interfaces, templates) and metadata decorators. The design goal is to let teams express domain shapes and service surfaces once, then emit many downstream artifacts (OpenAPI, Protobuf/gRPC, JSON Schema, SDKs, docs) from a single source of truth. This process is called an emitter pipeline.

A short TypeSpec example shows the basic authoring style and how decorators attach protocol semantics to a model:

```
model User {
  id: string;
  name: string;
  email?: string;
}
```

```
6.
7. @route("/users")
8. interface UsersService {
9.   @get list(): User[];
10.  @post create(@body user: User): User;
11. }
```

This snippet demonstrates three important TypeSpec ideas: first, *models* encode domain types; second, *interfaces* describe service operations; third, *decorators* (e.g., `@route`, `@get`, `@post`, `@body`) attach transport-level semantics that emitters use when producing HTTP/OpenAPI artifacts.

TypeSpec's emitter framework is central to its value proposition. Emitters are reusable libraries that "reflect" on the compiled TypeSpec model and generate textual artifacts; there are first-party and community emitters for OpenAPIv3, Protobuf (for gRPC), JSON Schema, client SDK scaffolds, and documentation. The emitter architecture provides composable building blocks so teams can write or extend emitters without reimplementing low-level formatting logic.

From a practical standpoint, TypeSpec is designed to be integrated directly into developer toolchains. There is an official VS Code extension that provides language services (syntax highlighting, completion, diagnostics) and the compiler (`tsp`) runs in CI to validate models and run emitters. Teams commonly place the TypeSpec compilation and emitter invocation inside build pipelines so that generated OpenAPI documents, Protobuf files, SDKs and docs are produced and published automatically on merge (Figure 2). The TypeSpec GitHub project and community guides emphasize this CI/CD orientation and show typical workflows for emitter invocation and automated publishing.

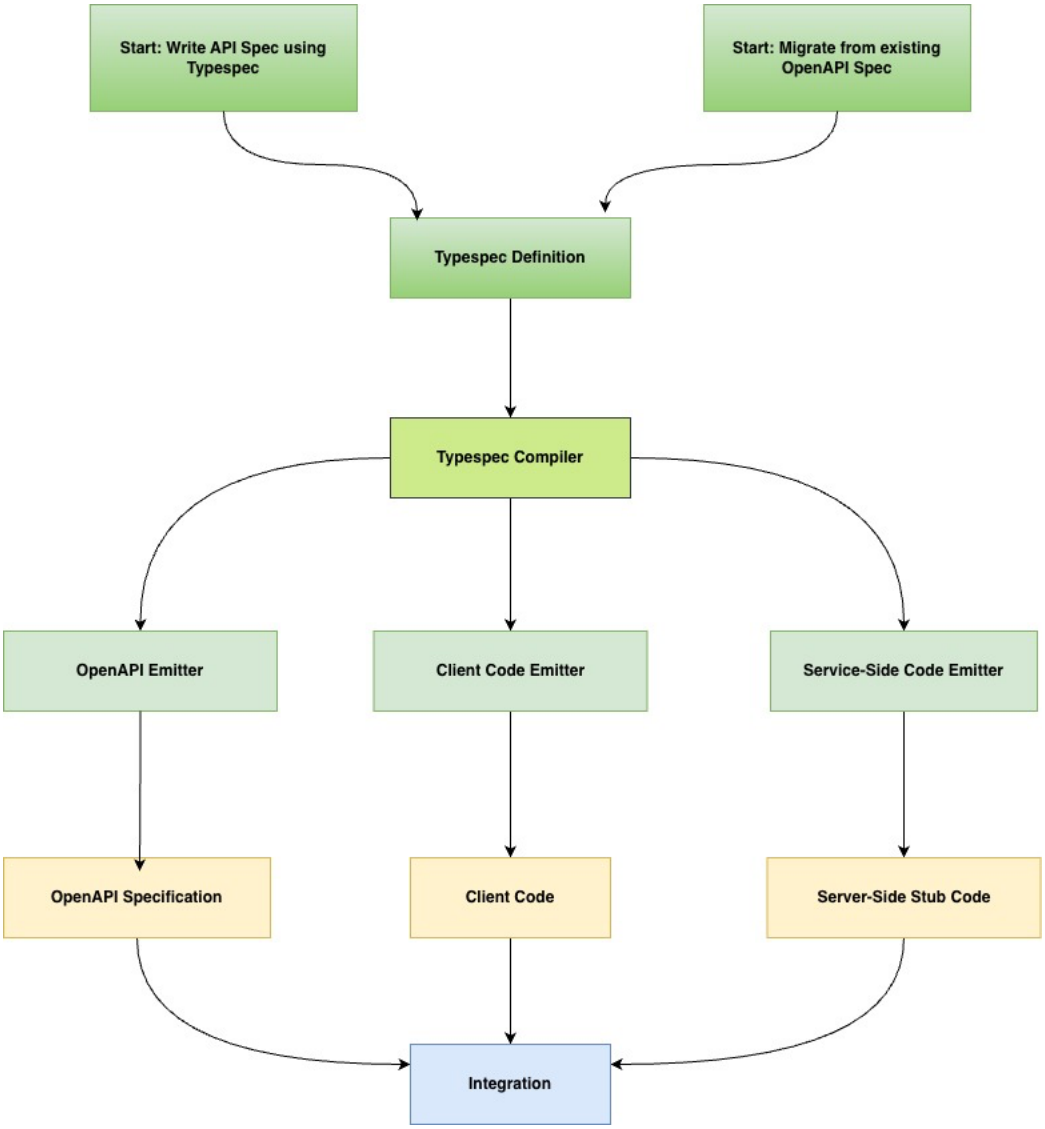


Figure 2 Typespec build pipelines

There are, however, practical caveats and mapping constraints to be aware of. When using TypeSpec to generate different kinds of outputs (like OpenAPI or Protobuf), there are some real-world limitations to keep in mind. Each output format has its own rules and capabilities: for example, Protobuf requires numbered fields and doesn't support every complex type feature that TypeSpec allows. Because of that, a TypeSpec model that looks perfect for an HTTP API might not automatically work well when converted to Protobuf.

So, when a team wants to generate multiple outputs from the same model, they have to be careful in how they design it. They might need to adjust parts of the model, simplify certain structures, or add extra hints (through decorators or annotations) that tell the generator how to handle specific cases.

In summary, TypeSpec is a pragmatic, extensible model-first system that emphasizes developer ergonomics, emitter-driven outputs, and organizational guardrails. It is particularly appealing

when teams need to generate multiple artifacts (OpenAPI for external docs, Protobuf for internal RPC, client SDKs for consumers) from a single, maintainable model and when they want to encode organizational policies as executable decorators and linters that run in CI. The tradeoffs are the need to understand emitter constraints and the current maturity of the ecosystem compared to older standards.

4 – The Importance of Model-Design-First in API Development Lifecycle and Collaboration

4.1 The API Development Lifecycle with Model-First Design

The API development lifecycle traditionally follows a sequence of stages including planning, design, implementation, testing, deployment, and maintenance. However, in many organizations, these stages occur in silos, leading to communication gaps, inconsistent specifications, and costly rework. The Model-First design paradigm redefines this lifecycle by establishing the API model as the single, authoritative artifact from which all downstream processes derive.

In a Model-First lifecycle, development begins with defining the API model which is an abstract, technology-neutral description of services, resources, and data contracts. Using modeling languages such as Protocol Buffers (for gRPC), Smithy, or TypeSpec, teams specify every endpoint, data type, and interaction pattern upfront.

Once the model is finalized and validated, automated code generation pipelines produce service stubs, client SDKs, and documentation directly from the model. This automation ensures synchronization between the specification and implementation, property that is a common pain point in code-first approaches, where documentation and SDKs often lag behind actual service changes. By integrating these steps into continuous integration workflows, Model-First frameworks enforce schema validation, version control, and compatibility checks before deployment.

A key differentiator of the Model-First lifecycle is that testing and validation begin at the design phase, rather than post-implementation. Mock servers or schema validators can simulate interactions defined by the model, allowing developers and testers to verify behavior early. This early testing ensures that inconsistencies or design flaws are identified before they propagate to production environments.

Furthermore, Model-First design facilitates API evolution and versioning. By maintaining a well-defined model repository, organizations can introduce non-breaking changes systematically by adding new fields, deprecating endpoints, or altering protocols while preserving backward compatibility for existing clients. This approach promotes evolutionary design, aligning with principles of sustainable architecture and agile delivery.

The lifecycle under Model-First design can thus be visualized as a closed feedback loop:

1. **Model Definition:** Specification of data structures, endpoints, and semantics.
2. **Review and Validation:** Cross-team collaboration and schema verification.
3. **Artifact Generation:** Automated production of SDKs, documentation, and tests.
4. **Implementation:** Development of business logic based on generated contracts.
5. **Deployment and Monitoring:** Continuous validation against model rules.
6. **Evolution:** Controlled schema updates and backward-compatible extensions.

This cyclical structure contrasts sharply with the linearity of code-first models. The Model-First approach ensures that design, implementation, and maintenance are interconnected, promoting consistency, traceability, and alignment between technical and business objectives. Ultimately, the Model-First API lifecycle establishes a foundation for scalable, automated, and predictable API **delivery**, reducing technical debt and enhancing organizational agility in rapidly changing environments.

4.2 Cross-Team Involvement and Non-Engineer Participation

One of the most transformative aspects of Model-First API design is its ability to enable cross-functional collaboration by bringing together engineers, architects, product managers, designers, and even non-technical stakeholders around a single, comprehensible model. This marks a fundamental departure from traditional code-first development, where design knowledge is often encoded directly in source code and thus inaccessible to non-engineers.

In large organizations or multi-team environments, APIs are more than technical constructs, they are interfaces between business capabilities. The Model-First paradigm provides a common language of collaboration, expressed through human-readable modeling specifications such as Smithy, TypeSpec, or Protocol Buffers. These languages describe service contracts, message formats, and endpoint behaviors in a declarative and structured way that can be easily reviewed, versioned, and discussed.

This shared model allows non-engineers to participate meaningfully in the design process. For instance:

- **Product managers** can validate that the exposed endpoints and operations align with user stories and business objectives.
- **UX designers** can ensure that API interactions are consistent with intended user flows or data consumption patterns.
- **Legal and compliance officers** can review data handling and exposure policies at the contract level.
- **Technical writers** can generate and refine documentation directly from the evolving model, keeping it synchronized with design changes.

Such collaboration is facilitated by machine-readable specifications that can be visualized or exported into user-friendly tools including API explorers, documentation portals, and schema visualization dashboards. This participatory design approach fosters early alignment between business and technical goals. Instead of downstream reviews after code completion, non-developers collaborate before implementation, reducing miscommunication and costly rework. The result is an API that is not only technically sound but also fit for purpose, accurately representing the business domain it serves.

Additionally, Model-First workflows empower organizations to adopt Design Reviews as a Service by utilizing automated pipelines that validate new or modified models against internal standards, governance rules, and organizational best practices. This enables distributed teams to

contribute independently while maintaining global consistency, even across large-scale, multi-cloud environments [24].

In summary, Model-First design promotes a culture of collaboration and transparency, bridging the gap between engineers and non-engineers. It transforms API design from a technical implementation detail into a shared organizational asset, fostering a collective understanding of the system's purpose, constraints, and evolution.

4.3 – Patterns, Consistency, and Governance

One of the key advantages of adopting a Model-First approach in API development is the ability to enforce patterns and consistency across services, supported by well-defined governance mechanisms. This section explores how shared modeling conventions, reusable design templates, and governance policies help ensure uniformity, maintainability, and compliance across the API landscape.

1. Pattern-Driven API Design

Model-First design facilitates the establishment of reusable design patterns that encode best practices across an organization. These patterns such as standardized pagination, authentication, error handling, or naming conventions can be defined once at the model layer and automatically inherited by every service definition. For instance, a Smithy or TypeSpec model may include shared traits or decorators for HTTP behaviors, versioning, or data validation, allowing teams to maintain consistent API behavior across microservices.

Such modeling standards reduce redundancy and prevent the "reinvention of the wheel" by providing a blueprint for service design, ensuring that new APIs follow consistent design principles aligned with business and technical objectives. Reusable interface models, standardized type libraries, and shared schemas enhance the coherence of an organization's overall API portfolio.

2. Consistency as a Quality Enabler

Consistency in APIs directly influences developer experience, integration simplicity, and long-term maintainability. When naming conventions, status codes, and data structures remain uniform, developers can easily navigate between services without additional learning curves. Model-First frameworks promote this uniformity by treating the API model as the single source of truth from which documentation, SDKs, mocks, and test cases are derived automatically. This tight alignment between design and implementation minimizes discrepancies and reduces the risk of drift between specification and deployed behavior.

Consistency also extends to semantic and structural design. For example, if two teams expose a "Customer" resource, governance rules can enforce schema alignment or type inheritance to maintain semantic integrity. These consistency layers make APIs easier to consume, reducing friction for both internal developers and external partners.

3. Governance Through Centralized Models and Validation

Governance in a Model-First ecosystem operates at the specification and validation level rather than through manual enforcement. By managing models in a central repository or registry (such as Stoplight, Postman, or internal Git-based systems), organizations can implement automated checks for compliance with internal guidelines, legal constraints, and regulatory frameworks [12].

Automated governance pipelines often built into CI/CD workflows validate each model against:

- **Organizational design rules** (naming, versioning, parameter consistency),
- **Security standards** (mandatory authentication schemes, encryption policies),
- **Interoperability norms** (JSON Schema or OpenAPI compatibility).

These validations help maintain API discipline at scale, ensuring that teams can innovate rapidly without compromising alignment or compliance. Model registries may also version and track schema evolution, enabling traceable changes and consistent backward compatibility management.

4. Governance as a Continuous Process

Finally, API governance in a Model-First environment is not static and it evolves with the organization's technology landscape and business priorities. As new standards (like GraphQL federation, AsyncAPI, or REST hooks) emerge, model validators and shared libraries must evolve accordingly. This adaptive governance ensures that the ecosystem remains future-proof and interoperable, while preserving the clarity and consistency that make Model-First methodologies effective at scale.

4.4 Model-First API Design in the Cloud Ecosystem

The growing adoption of cloud-native and multi-cloud architectures has transformed APIs from simple communication interfaces into the foundational integration layer of modern digital systems. Within this landscape, the Model-First API Design paradigm provides the structural and semantic discipline needed to manage APIs as scalable, interoperable, and governed assets across heterogeneous platforms.

Traditional code-centric development often produces fragmented API portfolios where each service evolving independently, tied to the specifics of its deployment environment. Model-First design eliminates these inconsistencies by defining APIs declaratively and independently of any single technology stack. The formal model, expressed in languages such as Smithy, TypeSpec, or Protocol Buffers becomes a cloud-agnostic abstraction that drives every downstream process: code generation, deployment configuration, documentation, security enforcement, and versioning.

By describing services and data structures through formal schemas, Model-First APIs align naturally with the declarative principles of cloud infrastructure. Just as Infrastructure-as-Code and Configuration-as-Code make cloud resources reproducible, Model-First design makes the interface itself reproducible. The API specification can be version-controlled alongside Terraform modules, Kubernetes manifests, or CI/CD pipelines, ensuring that every environment from

development to production implements the same canonical contract. This co-evolution of infrastructure and interface minimizes configuration drift and enables contract-driven deployments, where schema validation, backward-compatibility checks, and documentation generation occur automatically before release.

Major cloud platforms have embraced this approach. AWS API Gateway, Azure API Management, and Google Cloud Endpoints all natively import OpenAPI or gRPC definitions generated from Model-First frameworks, translating abstract models into fully configured runtime gateways. This automation bridges the gap between design and operations: when the model changes, the infrastructure updates itself accordingly. Security policies, throttling rules, and authentication flows defined at the model level, by using Smithy traits or TypeSpec decorators, propagate directly to the deployed service. In this way, governance and security cease to be post-deployment add-ons and become first-class design constructs, enforceable through automated pipelines rather than manual configuration.

Equally important is the role of Model-First design in achieving interoperability and portability across multiple clouds. Because the model defines only the semantics of communication services can emit multiple protocol representations (REST, gRPC, GraphQL) from the same specification. This allows the same logical API to function seamlessly across different cloud providers and runtime environments without rewriting its interface logic. Such schema-level portability protects organizations from vendor lock-in and supports hybrid architectures in which microservices operate across diverse infrastructures while sharing a single semantic vocabulary.

Automation also extends to the DevOps toolchain. Once an API model is committed to a repository, automated workflows can generate SDKs, mock servers, documentation, and gateway configurations, then validate them during CI/CD execution. Every pull request can trigger checks for breaking changes or non-compliant patterns before deployment, ensuring governance at scale. These pipelines make the API lifecycle continuous and self-correcting where new versions are validated, published, and monitored with minimal human intervention.

In essence, Model-First API Design operationalizes the principle of “Interface-as-Architecture.” It extends the declarative mindset of Infrastructure-as-Code to the design and governance of communication contracts themselves. By unifying modeling, automation, and governance, organizations gain the ability to deploy, evolve, and observe APIs coherently across any cloud. This synthesis of specification and execution marks a decisive step toward cloud-agnostic interoperability and sustainable API ecosystems.

4.5 The Role of Documentation in Model-First API Design

Documentation occupies a central position in the API development lifecycle, acting as the primary conduit through which design intent, technical specifications, and business semantics are communicated. In traditional code-first approaches, documentation is often considered an auxiliary output which is written after implementation or automatically extracted from source code

comments. This reactive treatment leads to frequent inconsistencies between the actual system behavior and its described interface, particularly in fast-moving development environments.

4.5.1 From Reactive Description to Proactive Design Artifact

Model-First API Design reverses this paradigm by embedding documentation at the core of the design process rather than positioning it at the periphery. The API model, expressed in a formal modeling language such as Smithy, TypeSpec, or Protocol Buffers, serves not only as an executable contract between producers and consumers but also as a continuously synchronized documentation source of truth.

Every structural element of the model, endpoints, parameters, schemas, and error types can be annotated with semantic metadata, descriptive text, examples, and constraints. These annotations are preserved in the compiled outputs (e.g., OpenAPI 3.1, JSON Schema, or SDK comments), allowing documentation to evolve in lockstep with the specification. As a result, documentation ceases to be a static text file and becomes a dynamic, machine-generated reflection of the model's formal semantics.

This structural integration addresses the two chronic weaknesses of code-first documentation: latency and divergence. Because documentation is generated from the authoritative model at build time, the risk of drift between implementation and description is effectively eliminated. Updates to the model, whether adding a new endpoint or deprecating a field, are immediately reflected in all derived documentation artifacts, ensuring accuracy and timeliness.

4.5.2 Mechanisms of Automated Documentation Generation

Each major Model-First framework provides tooling to generate comprehensive, navigable documentation directly from the model source:

- **Smithy** integrates traits such as `@documentation`, `@example`, and custom metadata tags, which propagate through the build pipeline to produce rich HTML or OpenAPI documentation. These outputs can include semantic relationships, inheritance hierarchies, and cross-references between services and data types.
- **TypeSpec** leverages decorators and scalar annotations to embed domain-specific semantics within the schema. Its compiler can emit both developer-friendly portals and machine-readable outputs, ensuring that documentation and schema validation share a common source.
- **gRPC/Protobuf**, through extensions like `google/api/annotations.proto`, supports automatic generation of OpenAPI specifications and API reference material that map RPC methods to REST-style endpoints, aligning transport-agnostic service definitions with human-readable documentation.

In all cases, documentation generation is not an optional auxiliary step but an integral part of the continuous integration pipeline. Whenever a new commit modifies the model, the build system re-emits the documentation, SDKs, and schema validators simultaneously. This creates a self-

sustaining documentation ecosystem, where accuracy, versioning, and discoverability are automated rather than manually enforced.

4.5.3 The Epistemic and Strategic Role of Documentation

From an epistemological perspective, Model-First documentation represents a codified form of organizational knowledge meaning that it describes a living artifact that records not just technical structures but also the rationale behind design decisions. Because it is versioned alongside the model, it allows engineers to reconstruct historical design contexts, compare schema revisions, and reason about the evolution of business capabilities over time. This traceability strengthens architectural governance and supports continuous auditing of service evolution.

Strategically, automated and semantically rich documentation enhances developer experience (DX) and accelerates ecosystem growth. Well-structured, always-synchronized documentation reduces onboarding time, increases adoption rates for public APIs, and improves integration reliability. It also provides a tangible metric for maturity within an organization's API governance framework, demonstrating how design consistency and transparency translate directly into operational efficiency.

5 – Static Analysis of Inter-Endpoint Dependencies

While documentation formats such as OpenAPI or Postman Collections describe each endpoint in detail, they rarely capture how endpoints depend on one another that is, how data produced by one API call becomes an input to another.

Understanding these inter-endpoint dependencies is essential for integration, testing, and evolution of complex systems. It allows developers to determine call order, recognize data flow, and identify potential reuse opportunities.

This chapter extends the concept of static dependency analysis into Model-First API Design frameworks, Smithy, TypeSpec, and gRPC / Protobuf, where schemas and operations are defined with strong typing and semantic structure. These models enable dependency discovery that is both accurate and structurally meaningful.

5.1 Inter-Endpoint Dependencies

An inter-endpoint dependency exists when an endpoint E_2requires data produced by another endpoint E_1. Formally, endpoint E_2depends on $E_1$when one or more fields in E_1's response are necessary to construct E_2's request.

Typical dependency types include:

- **Body → Path:** A response attribute is used as a path parameter in another request.
- **Body → Body:** A response field is reused within another request body.
- **Body → Query:** A response field is used as a query parameter.

5.2 Smithy Analysis

REST Endpoint Modeling with smithy.api#http

For this work, REST-style endpoints were modeled by applying the `smithy.api#http` trait to Smithy operations. This trait allows each operation to be annotated with an HTTP method (e.g., `GET`, `POST`, `PUT`) and a URI path pattern, effectively defining the REST interface in a declarative manner. Through these annotations, the Smithy model encodes complete endpoint semantics like paths, query parameters, and request/response bodies, within the abstract model itself, without requiring code generation.

Strongly-Typed I/O Definitions

A central strength of Smithy in this context lies in its ability to *wrap* primitive values inside structured shapes, thus expressing semantically meaningful data types. For example, two strings can be separated with the following syntax in smithy:

```
1. string Name
2. string Name2
```

Then we can use Name and Name2 in different occasions. This design enables non-primitive modeling of request and response data, ensuring that every input and output value carries both syntactic type and domain-level meaning.

Smithy Build and AST Extraction

Once the Smithy model is authored, it is compiled using the smithy build command, which generates a comprehensive Abstract Syntax Tree (AST) in JSON format. The AST contains every operation, shape, trait, and reference, preserving the complete semantic hierarchy of the service model. Unlike the OpenAPI emitter, which flattens structures and loses the wrapping of non-primitive types, the Smithy AST preserves these wrappers, maintaining the distinction between higher-level domain types and their underlying primitive values. This makes the AST the ideal intermediate representation for performing inter-endpoint dependency analysis.

Inter-Endpoint Dependency Analysis

With the Smithy AST as the input, the dependency analysis process seeks to uncover how one endpoint's output feeds into another's input. In essence, this analysis identifies data flow dependencies between operations by tracing shared shapes across their input and output hierarchies.

Algorithmic Process

The following algorithm outlines the procedure for computing these inter-endpoint dependencies using the Smithy AST:

```
1. Algorithm: Smithy Dependency Analyzer
2.
3. Input:  Smithy AST model (JSON)
4. Output: Directed graph of inter-endpoint dependencies
5.
6. 1. Parse the Smithy AST and extract all operations:
7.       - Operation name, HTTP method, input shape, output shape.
8.
9. 2. For each operation (source):
10.      a. Recursively collect all type identifiers (shapes) from its output.
11.      b. Record parameter traits (path, query, body) where applicable.
12.
13. 3. For every other operation (target):
14.      a. Recursively collect all type identifiers from its input and parameters.
15.      b. Compare each type in the source output with each in the target input.
16.      c. If a shared type is found:
17.             Record a dependency (source → target)
18.             Include shared type and parameter mapping.
19.
20. 4. Eliminate duplicate matches of the same type per operation pair.
21.
22. 5. Optionally restrict analysis to GET operations as sources.
23.
24. 6. Generate the dependency graph:
25.      - Nodes: operations
26.      - Edges: dependencies labeled with shared types.
```

5.3 TypeSpec Analysis

REST Endpoint Modeling with @typespec/http

REST-style endpoints were modeled in TypeSpec using the `@typespec/http` library, which provides decorators such as `@get`, `@post`, `@put`, and `@route` to declare REST operations. These decorators allow the full definition of HTTP methods, URI paths, and parameter locations (path, query, or body) directly within the model. As a result, the REST interface is described declaratively, and all endpoint semantics (methods, paths, and payload structures) are explicitly encoded within the TypeSpec source itself.

Strongly-Typed I/O Definitions

TypeSpec, like Smithy, supports wrapping primitive types with higher-level, domain-specific scalars. For instance, a developer can define:

```
1. scalar Name extends string;
2. scalar Name2 extends string;
```

These scalar definitions allow non-primitive modeling of API inputs and outputs, ensuring that identical base types (like `string`) can be semantically distinguished by their context. This strong typing guarantees that request and response data maintain domain meaning and enables the identification of dependencies between endpoints based on shared, named scalar or structured types.

TypeSpec Compilation and OpenAPI Generation

Once the TypeSpec model is authored, it is compiled using the `tsp compile` command with the `@typespec/openapi3` emitter to produce an OpenAPI specification. Unlike Smithy, TypeSpec does not expose a standalone AST format. However, its emitted OpenAPI retains type references for wrapped primitives, represented as schema references such as:

```
1.  $ref: '#/components/schemas/Name'
2.  Name:
3.  type: string
```

This means that the TypeSpec emitter preserves these scalar wrappers. As a result, type-level distinctions remain visible in the exported OpenAPI, allowing accurate and fine-grained dependency analysis directly at the schema level.

Inter-Endpoint Dependency Analysis

Using the OpenAPI document generated from TypeSpec as input, the inter-endpoint dependency analysis follows the same conceptual framework as in Smithy. The goal is to identify how one endpoint's response data (source) provides input to another endpoint (target). This is accomplished by tracing shared schema references across the request and response structures within the OpenAPI document.

Algorithmic Process

The procedure for detecting dependencies is outlined below:

```
1. Algorithm: TypeSpec Dependency Analyzer
2.
3. Input:  OpenAPI specification generated from TypeSpec
4. Output: Directed graph of inter-endpoint dependencies
5.
6. 1. Parse the OpenAPI specification.
7. 2. Extract all operations:
8.       - HTTP method, path, input and output schema references.
9.
10. 3. For each operation (source):
11.       a. Collect all schema references ($ref) from its response body.
12.
13. 4. For every other operation (target):
14.       a. Collect schema references from its request body and parameters.
15.       b. If a shared reference is found in both:
16.              Record a dependency (source → target)
17.              Include the shared schema and parameter context.
18.
19. 5. Skip duplicate matches for the same type between identical operation pairs.
20.
21. 6. Optionally restrict analysis to GET operations as dependency sources.
22.
23. 7. Generate a dependency graph:
24.       - Nodes: operations
25.       - Edges: dependencies labeled by shared schema references.
```

5.4 gRPC / Protobuf Analysis

RPC Modeling with Protocol Buffers

Unlike Smithy or TypeSpec, which model REST-style resources, Protobuf structures APIs around *services* and *RPC methods*. Each RPC defines a `request` and `response` message type, representing the input and output of a remote call.

For example:

```
1. service OrderService {
2.   rpc CreateOrder (CreateOrderRequest) returns (CreateOrderResponse);
3.   rpc GetOrder (GetOrderRequest) returns (Order);
4. }
```

This structure emphasizes procedural semantics through method calls instead of resource manipulations, while maintaining a strong type system using message definitions.

REST Mapping with google/api/annotations.proto

To enable REST-style interoperability, gRPC services can be annotated with the `google/api/annotations.proto` extension. This allows RPCs to be mapped directly to HTTP endpoints via the `google.api.http` option, defining REST semantics such as HTTP methods, paths, and parameter locations. For example:

```
1. rpc GetOrder (GetOrderRequest) returns (Order) {
```

```
    option (google.api.http) = {
      get: "/v1/orders/{order_id}"
    };
}
```

Through these annotations, gRPC services can be exported as OpenAPI specifications, making them compatible with the same dependency analysis pipeline used for Smithy and TypeSpec.

Type Modeling and Limitations

While Protobuf is strongly typed, it differs from Smithy and TypeSpec in one key respect: it does **not** support wrapping primitive types into higher-level domain scalars. Each primitive field such as `string`, `int32`, or `bool` is defined directly within messages, without an alias or domain-level identifier. Consequently, when converting gRPC services to OpenAPI, only structured message types (component schemas) retain their identities, while primitive fields lose their contextual meaning. This makes dependency inference based purely on primitives more ambiguous.

The OpenAPI specification used for the analysis is generated directly from the `.proto` definitions using the Protocol Buffers compiler (`protoc`) with the OpenAPI plugin, as shown below:

```
protoc \
  --openapi_out=out \
  --proto_path=. \
  my_service.proto
```

This command compiles the Protobuf service definitions into an OpenAPI document (`.yaml` or `.json`), preserving all structured message types and HTTP annotations (via `google/api/annotations.proto`).

To mitigate this, the analysis operates under two modes:

1. **Component Mode** – Compares only structured message references, producing high-precision dependencies between RPCs that share message types.
2. **Primitive Mode** – Broadens analysis by also comparing primitive fields (by name and type). This captures additional potential dependencies but may introduce false positives.

Inter-RPC Dependency Analysis

The dependency analysis for gRPC follows the same conceptual structure as for Smithy and TypeSpec, with the OpenAPI export acting as the input. The goal remains to identify how data produced by one RPC can serve as input to another. In this context, an RPC's response message type acts as the *source*, while another RPC's request message type serves as the *target*. If they share a common structured message or field, a dependency edge is created.

Algorithmic Process

The procedure for detecting dependencies is outlined below:

```
Algorithm: gRPC / Protobuf Dependency Analyzer

Input:  OpenAPI specification generated from .proto files
```

```
Output: Directed graph of inter-endpoint dependencies

1. Parse all operations and extract:
    - RPC name, HTTP method, input/output message schemas, parameters.

2. For each operation (source):
    a. Recursively collect all schema references ($ref) and primitive fields
       from its response body.

3. For each other operation (target):
    a. Recursively collect schema references and primitive fields from
       its request body and parameters.
    b. Compare:
        i.  Schema references → record if identical (Component Mode)
        ii. Primitive name/type pairs → record if matching (Primitive Mode)
    c. Label each dependency with the reason ("ref" or "primitive").

4. Deduplicate identical matches per operation pair.

5. Optionally restrict analysis to GET operations as sources.

6. Generate the dependency graph:
    - Nodes: RPC endpoints
    - Edges: dependencies labeled by match reason and type.
```

The next chapter presents a PayPal API case study, comparing the dependency graphs generated by RADAR, Smithy, TypeSpec, and gRPC analyzers. This evaluation quantifies improvements in accuracy and demonstrates the advantages of model-first representations for dependency discovery.

6 – Case Study: PayPal API Dependency Analysis

To validate the proposed methodology for endpoint interdependency detection, a real-world API scenario was selected as a case study. The use case originates from the previous diploma thesis about dynamic analysis [8], [9].

6.1 Use Case Definition

The analyzed workflow, originally defined in the thesis as Use Case 1: New Order, Payment, and Tracking Information, simulates the complete lifecycle of an e-commerce transaction using the PayPal REST API. It was chosen because it embodies realistic interdependencies between endpoints across multiple subsystems including catalog management, checkout processing, payment authorization, and shipment tracking.

The workflow begins with the creation of a new product in the PayPal catalog. Once created, its details are retrieved and updated to modify descriptive fields. Next, an order is placed for the same product. The shipping address associated with that order is updated, and the order's information is subsequently retrieved for confirmation. The next phase focuses on financial transactions: the payment of the order is first authorized, confirming that the customer's balance is sufficient, and is then captured by the seller. Once captured, the payment details are retrieved. Finally, tracking information is added to the previously captured payment, and the record is updated. This workflow is also presented on the following Sequence diagram.

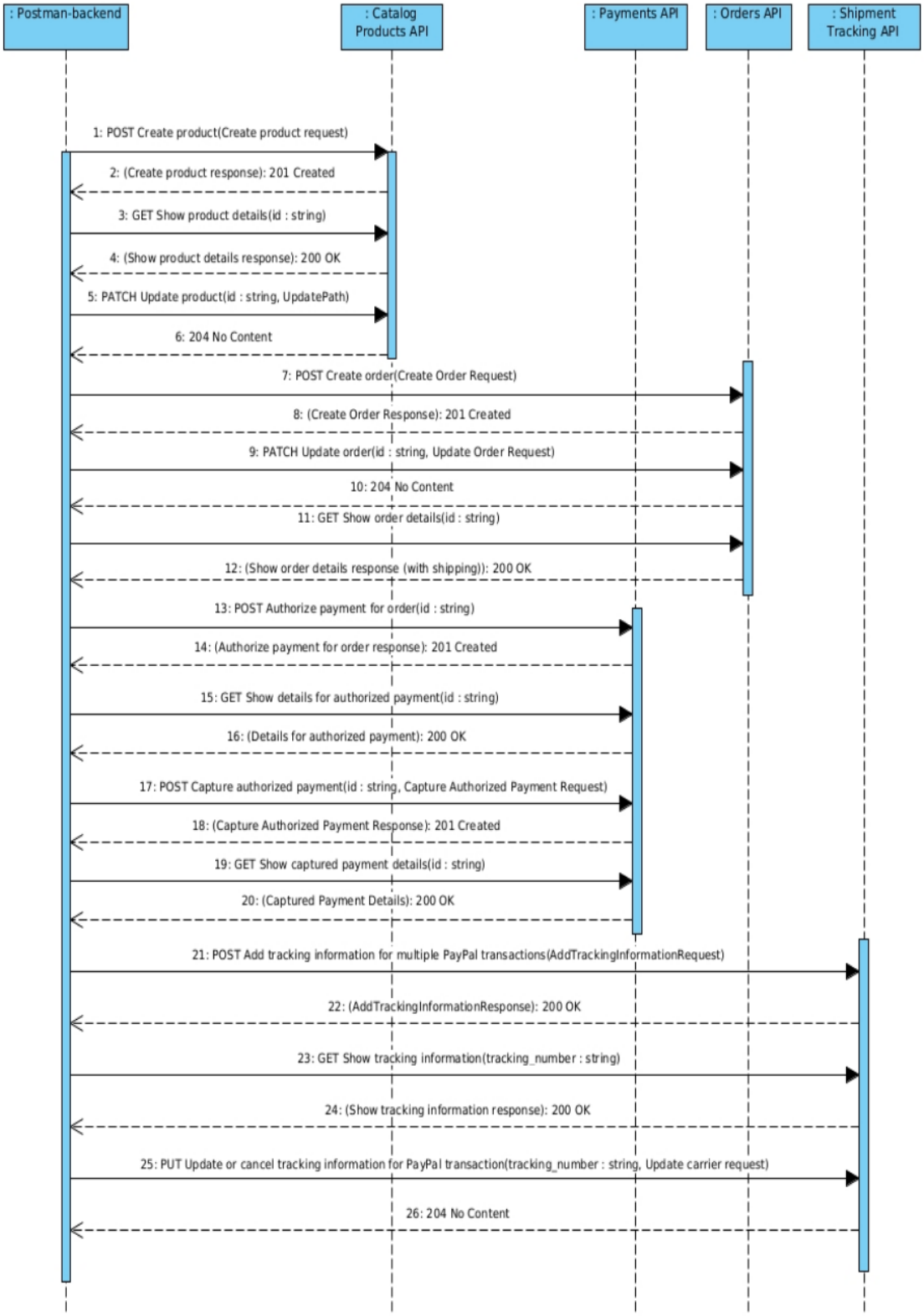


Figure 3 PayPal Use Case 1: sequence diagram

This workflow encapsulates multiple interdependent entities such as Product_Id, Order_Id, Payment_Id, and Tracking_Id. Each identifier is generated or returned by one endpoint and later consumed by another, forming a chain of logical dependencies that reflect the real data flow within the PayPal API.

6.2 Model Definition

To perform the comparative evaluation, the PayPal API workflow was implemented using three different Model-First API frameworks: Smithy, TypeSpec, and gRPC/Protobuf. In all three, the same endpoints were modeled with equivalent names, paths, and data structures, ensuring consistency across frameworks. The implemented endpoints included:

- `POST /v2/checkout/orders` – Create an order
- `GET /v2/checkout/orders/{order_id}` – Show order details
- `PATCH /v2/checkout/orders/{order_id}` – Update an order
- `POST /v2/checkout/orders/{order_id}/authorize` – Authorize payment for order
- `GET /v2/payments/authorizations/{authorization_id}`- Show authorized payment details
- `POST /v2/payments/authorizations/{authorization_id}/capture` – Capture an authorized payment
- `GET /v2/payments/captures/{capture_id}` - Show captured payment details
- `POST /v1/catalogs/products` – Create a product
- `GET /v1/catalogs/products/{product_id}` – Show product details
- `PATCH /v1/catalogs/products/{product_id}` – Update a product
- `POST /v1/shipping/trackers-batch` – Add tracking information for multiple PayPal transactions
- `PUT /v1/shipping/trackers/{tracking_id}` – Update or cancel tracking information for a specific PayPal transaction

Each of these endpoints was encoded using the respective modeling language constructs of the framework. Smithy used `@http` traits and input/output structures; TypeSpec used annotated interfaces and scalars; and gRPC defined messages and RPC methods, annotated with `google/api/annotations.proto` to export REST-compatible OpenAPI definitions. This cross-framework implementation provided a controlled environment for assessing how each modeling paradigm affects dependency detection accuracy.

6.3 Dependency Extraction Procedure

The dependency extraction process was carried out by the analyzers implemented in Chapter 5, each adapted to handle the corresponding model format. The process follows the same general logic across frameworks: for each operation, the analyzer extracts its input and output schemas and recursively collects all referenced types. It then compares the output types of each operation

with the input types of every other operation. When a shared type or reference is found, a directed dependency is recorded from the producer (source endpoint) to the consumer (target endpoint).

6.3.1 Smithy Analysis

For Smithy, the analysis utilized the AST (Abstract Syntax Tree) generated by the Smithy build system. In this model, all primitives were defined as semantic wrappers, for example:

```
1. string Order_Id
2. string Product_Id
```

These were preserved in the AST as fully qualified type identifiers such as `smithy.paypal#Order_Id`, enabling the analyzer to distinguish between conceptually different fields that would otherwise share the same primitive type. This design allowed the dependency analyzer to capture relationships based on type identity rather than mere textual similarity.

6.3.2 TypeSpec Analysis

The TypeSpec implementation followed the same principle. Semantic primitives were represented as scalar extensions, for example:

```
1. scalar Order_Id extends string;
2. scalar Product_Id extends string;
```

When compiled to OpenAPI, TypeSpec preserved these scalar types as `$ref` entries in the `#/components/schemas` section. This ensured that type identity was maintained throughout the export process, enabling the analyzer to match dependencies at the schema level. The TypeSpec analysis produced results identical to those of Smithy: all dependencies were correctly identified, with high-level objects and wrapped primitives both contributing to meaningful connections. Complex types such as `Amount`, `Payment`, and `Tracker` were treated as unified entities rather than fragmented sets of fields, reinforcing the semantic integrity of the graph.

6.3.3 gRPC / Protobuf Analysis

For the gRPC implementation, `.proto` files were compiled to OpenAPI using the `protoc`.

This generated OpenAPI specifications that exposed Protobuf message structures as JSON schema components. However, Protobuf does not support the concept of semantic primitive wrapping; thus, identifiers such as `string order_id` are represented only by their basic type. As a result, the analyzer could detect dependencies only by comparing field names and primitive types.

This introduced two issues: false dependencies appeared when different resources shared common field names (e.g., `id`), and true dependencies were sometimes missed when naming differed (e.g., `order_id` versus `id`). Nevertheless, the analyzer successfully captured message-level dependencies, maintaining meaningful high-level relationships while losing some detail in primitive-level matching.

6.4 Comparative Findings

Both Smithy and TypeSpec produced virtually identical and highly accurate dependency graphs as it is presented on Figure 3 and 4. Their analyses captured all genuine data flow relationships within the PayPal endpoints and eliminated noise by using explicit type-based matching.

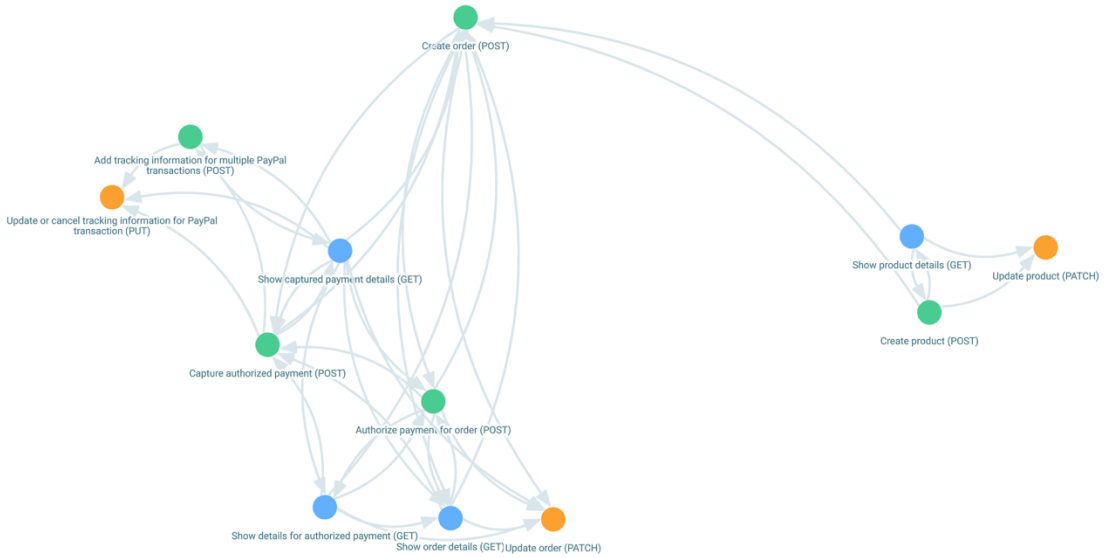


Figure 4 smithy complete graph

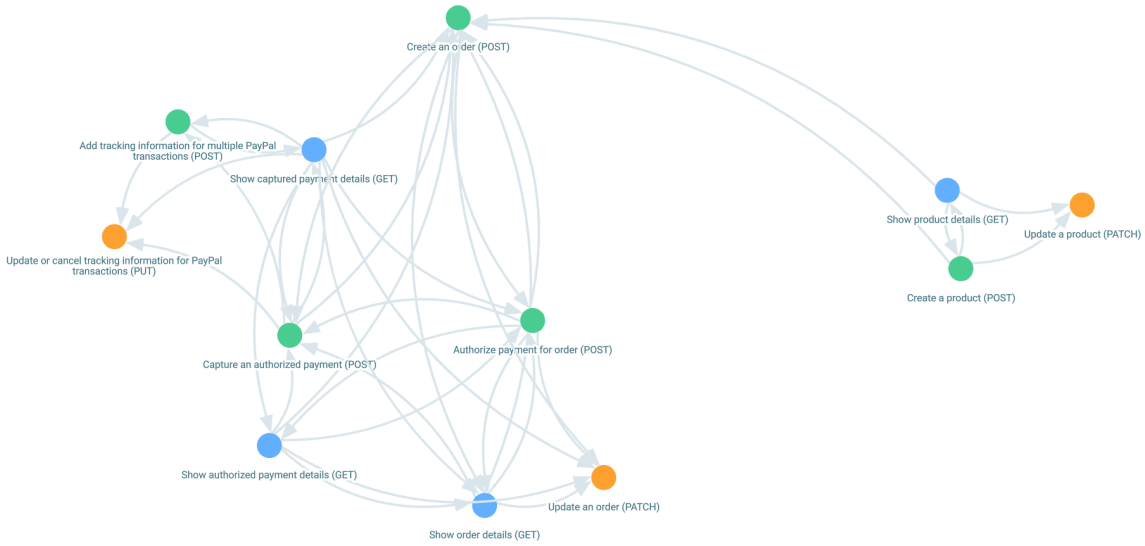


Figure 5 typespec complete graph

In contrast, the gRPC results exhibited less semantic cohesion due to the structural limitations of Protocol Buffers, which do not support primitive type wrapping. When the analysis was performed excluding primitive types, the algorithm successfully identified all high-level dependencies between complex objects, accurately reflecting the true inter-service relationships as shown in Figure 5.

However, when the primitive inclusion mode was enabled, where dependencies were also inferred based on matching primitive field names and types, the output became significantly noisier, as illustrated in Figure 6. The analyzer began linking fields that were textually similar but semantically unrelated, resulting in fragmented or misleading relationships. Moreover, because Protobuf does not distinguish between semantically different identifiers that share primitive types, it was unable to associate related fields with different names (for example, recognizing that id and orderId refer to the same logical entity). Consequently, the gRPC-based analysis, while capable of capturing object-level dependencies, lacked the semantic precision and abstraction depth achieved by Smithy and TypeSpec.

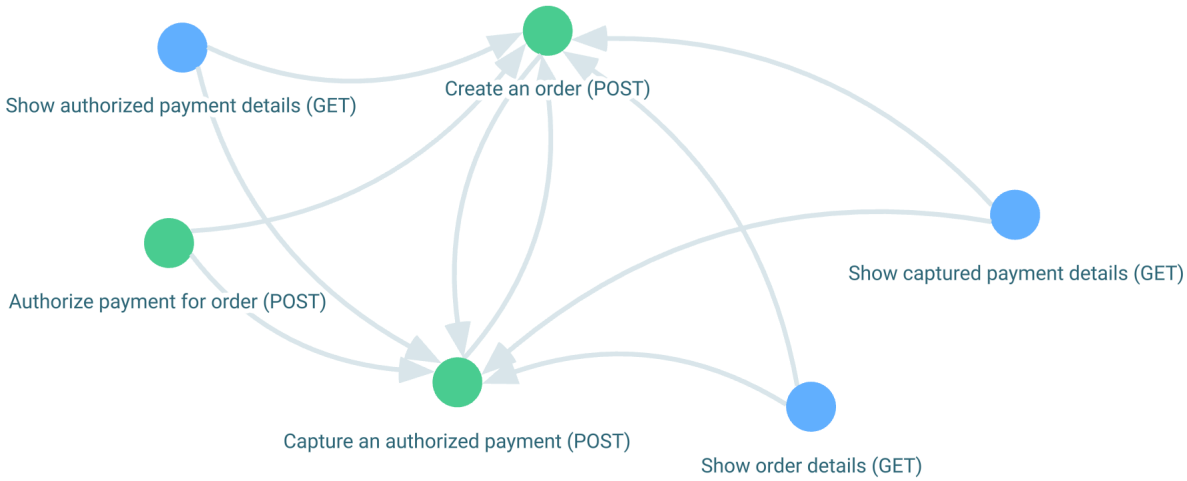


Figure 6 proto complete graph without primitives

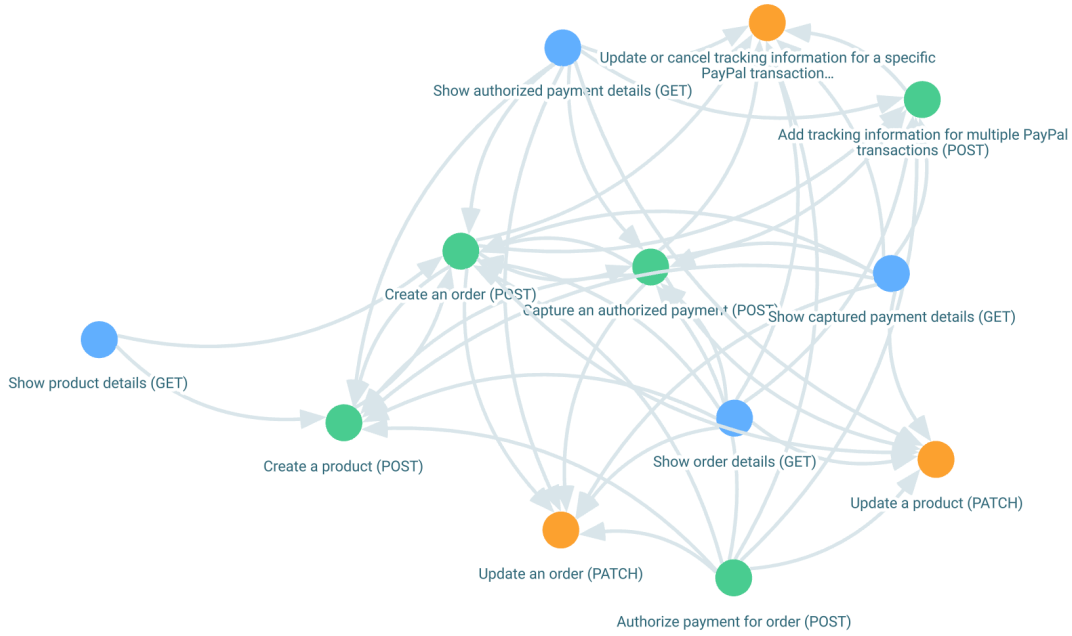


Figure 7 proto complete graph with_primitives

6.5 Comparison with existing RADAR analyses

To evaluate the effectiveness and accuracy of the proposed Model-First dependency analysis methodology, a direct comparison was conducted against both the static and dynamic analysis results of the RADAR software. For this comparison, the Smithy-based model was selected as the reference implementation, since its dependency extraction logic and results are identical to those obtained from TypeSpec, and it provides a more expressive and semantically consistent abstraction than the gRPC approach, which as discussed previously suffers from limitations in representing wrapped primitive types.

At first glance, when examining the complete dependency graph generated by the Smithy analyzer (Figure 3), it is evident that it contains more connections compared to both the static (Figure 7) and dynamic (Figure 8) RADAR results. This outcome was expected, as the Model-First methodology leverages the semantic precision of the Smithy Abstract Syntax Tree (AST) to match dependencies based on the formal type relationships defined in the model, rather than relying on approximate name or value matching. In contrast, the static analysis used in RADAR's original implementation tends to lose significant dependency information because it relies exclusively on textual similarity between names and primitive types, without the ability to infer higher-level type semantics. Consequently, the static graph appears sparser and omits several meaningful relationships that are structurally encoded in the Smithy model.

The dynamic analysis (Figure 8), although producing a more meaningful dependency graph than the static approach, captures dependencies only for interactions actually observed during the MIM-based execution tracing. As a result, it provides a partial view of the API's interconnectivity, accurate for the executed workflows but inherently limited to the recorded behavior. When focusing on specific endpoints, these differences become even clearer.

For example, in the case of the Create order endpoint, both the Smithy-based model and the dynamic analysis successfully identify all four real dependencies, as shown in Figures 9 and 11. In contrast, the static analysis (Figure 10) detects only one dependency, omitting critical relationships such as the link between `id` and `order_id`. This omission demonstrates the static analyzer's inability to recognize dependencies across fields that differ in name but share a semantic relationship defined in the model. In this scenario, the dynamic analysis performs well because it captures the actual runtime data flow of the use case, allowing it to show dependencies that the static approach misses.

However, when examining another endpoint, such as Create Product, the situation reverses. The Smithy-based dependency graph (*Figure 12*) accurately identifies a dependency between `CreateProduct` and `CreateOrder`, represented by the relationship `name → name` (through the type `smithy.paypal#Product_Name`). The static analysis also detects this dependency (*Figure 13*), since the field names are identical and thus easily matched. In contrast, the dynamic analysis (Figure 14) fails to identify this relationship entirely, as the corresponding workflow was not executed in the recorded MIM session. This highlights an important limitation of dynamic

methods: while they produce precise results for executed interactions, they cannot infer dependencies that have not been empirically observed.

Overall, the comparison demonstrates that the Smithy-based dependency analyzer combines the semantic precision of static analysis with the completeness of dynamic inference, without requiring execution data. It captures both explicit object-level dependencies (e.g., Amount, Product, Order) and implicit semantic relationships (e.g., `Order_Id`) by leveraging the model's structured type information. In contrast, RADAR's static approach loses many such dependencies due to its reliance on surface-level matching, while the dynamic approach, though behaviorally grounded, is constrained by the scope of the monitored use cases.

Consequently, the Model-First methodology achieves a more comprehensive, noise-free, and semantically rich dependency graph, enabling deeper insights into API interoperability and lifecycle coupling than either of RADAR's existing methods.

Figure 8 static analysis complete graph

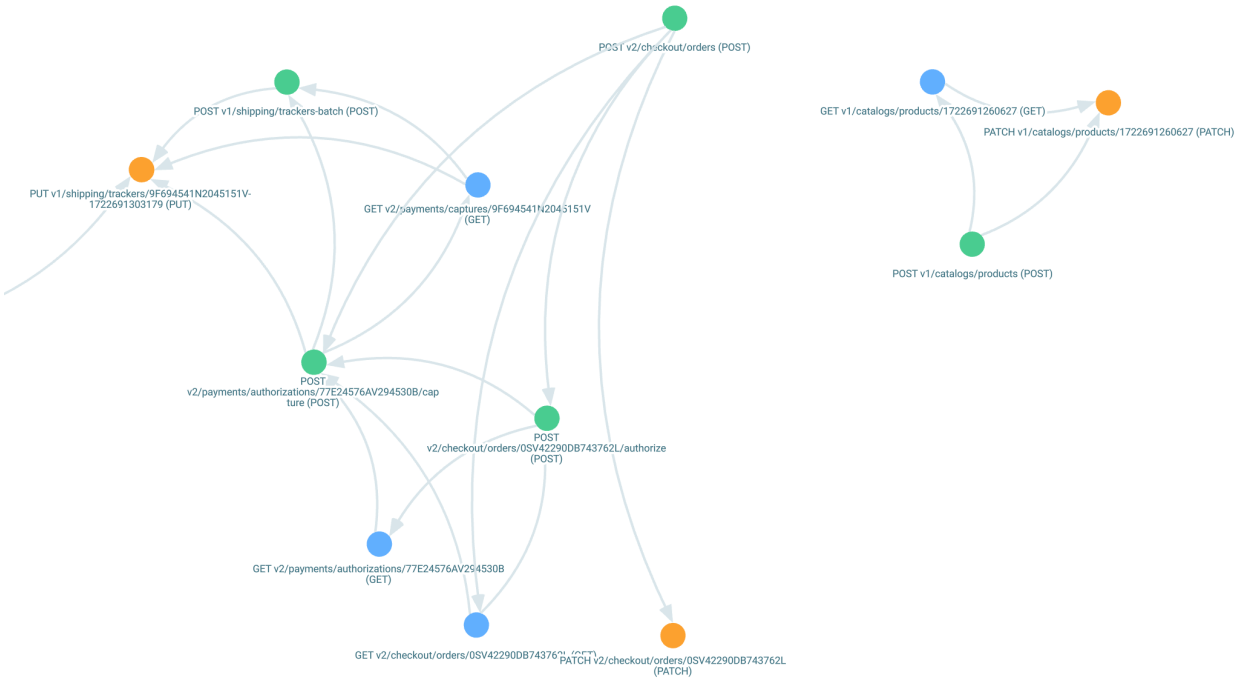


Figure 9 dynamic analysis complete graph

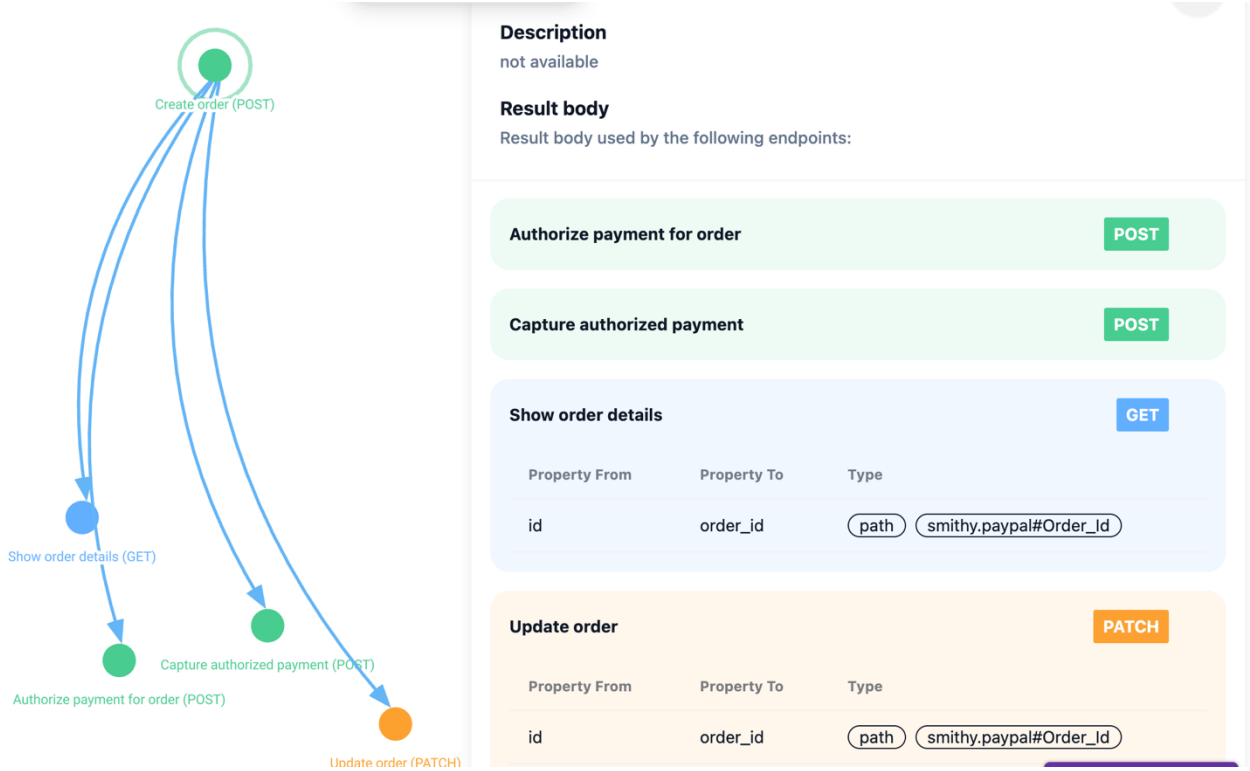


Figure 10 smithy create_order_dependencies

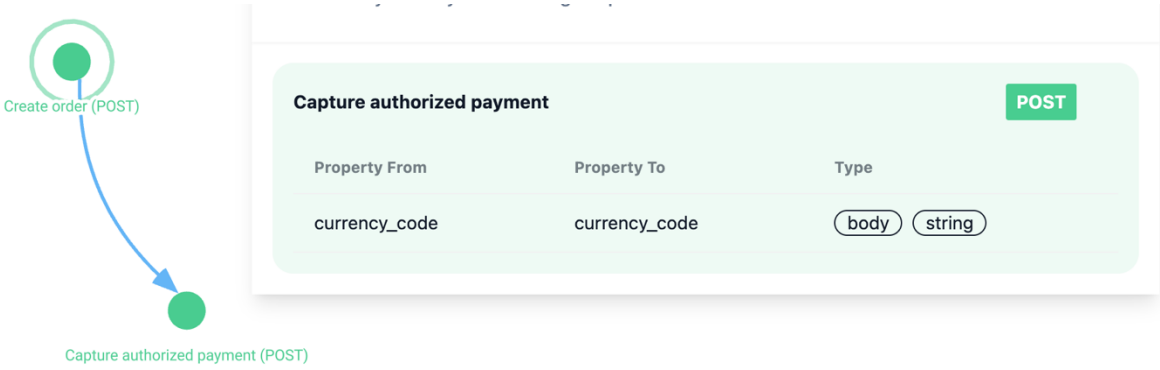


Figure 11 static analysis create_order_dependencies

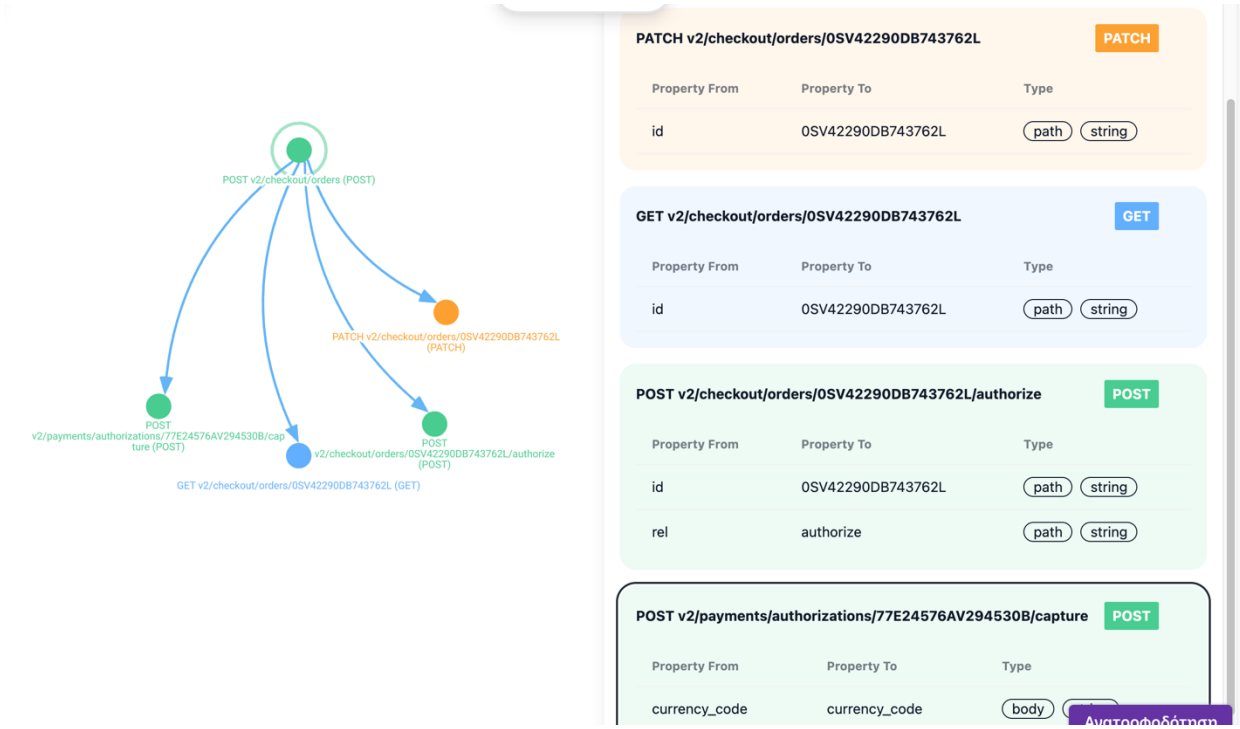


Figure 12 dynamic analysis create_order_dependencies

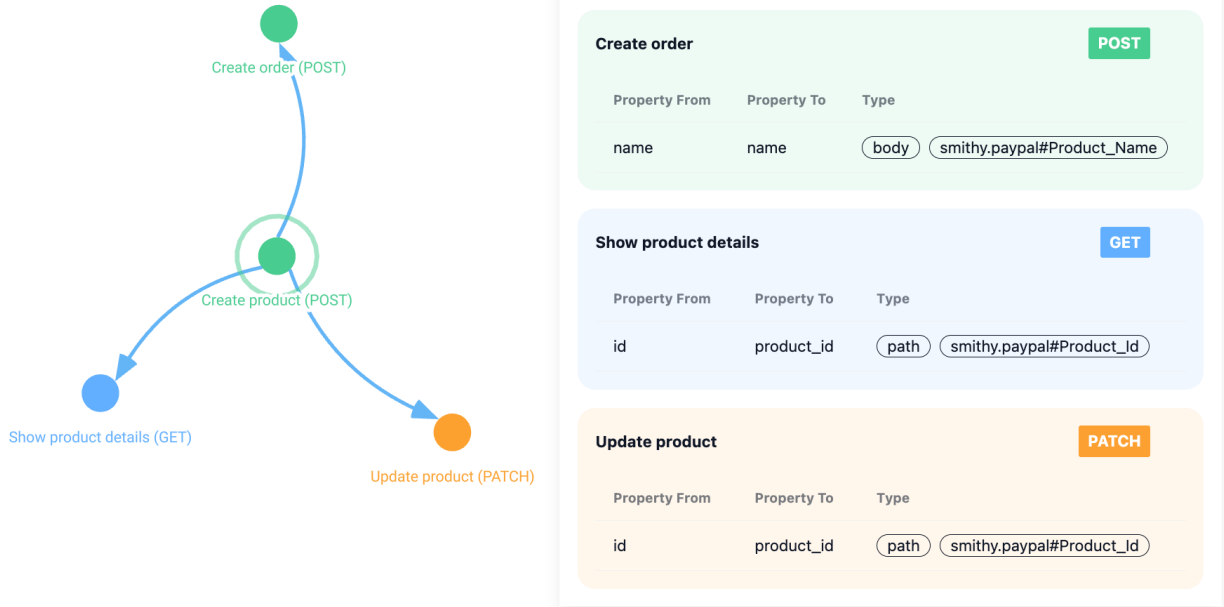


Figure 13 smithy analysis create_product_dependencies

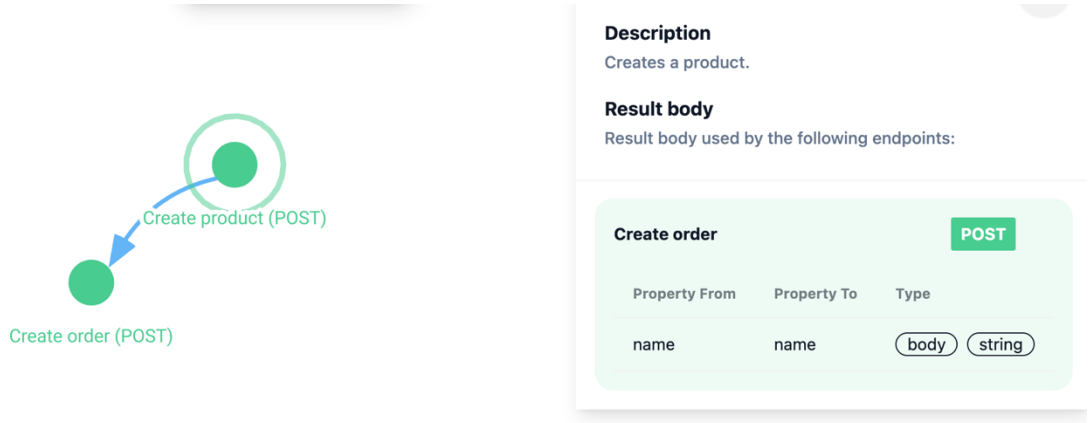


Figure 14 static analysis create_product_dependencies

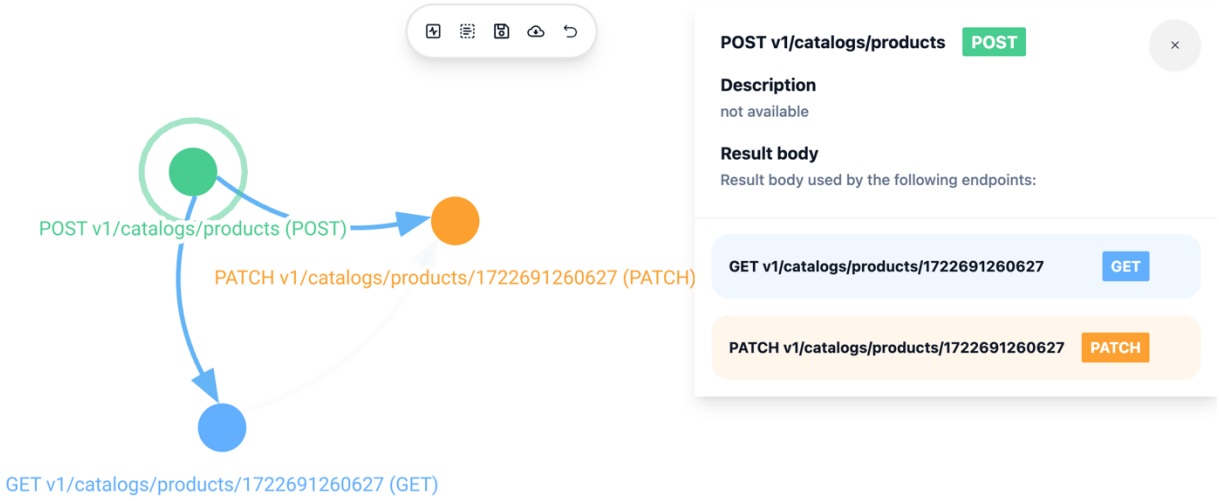


Figure 15 dynamic analysis create_product_dependencies

7 – Conclusion

In this thesis, we introduced and thoroughly analyzed the concept of Model-First API Design, a paradigm that redefines how APIs are planned, implemented, and maintained.

Three representative frameworks were examined in detail: gRPC (Protocol Buffers), Smithy, and TypeSpec. Each framework embodies the principles of model-first design but applies them within different technical and operational contexts. The analysis revealed that gRPC is the best option in low-latency, internal communication environments, where efficiency and binary serialization are paramount. However, it is less expressive in modeling domain semantics due to its inability to wrap primitive values which limits its potential for structural dependency analysis. In contrast, Smithy and TypeSpec proved to be more mature**,** language- and protocol-agnostic frameworks that emphasize extensibility, interoperability, and design consistency. Between them, Smithy currently offers a richer ecosystem and a more advanced trait system, while TypeSpec provides a modern, developer-friendly approach to specification generation and code integration.

Beyond technical differences, the study highlighted the broader organizational value of Model-First design. By defining APIs as formal models, teams gain a single source of truth that promotes cross-team collaboration, early validation, and automatic documentation generation. These qualities are essential in large-scale, cloud-native environments where multiple teams contribute to the same API ecosystem and where long-term maintainability is as critical as initial delivery.

A central contribution of this research lies in leveraging these modeling frameworks for inter-endpoint dependency analysis. Because Smithy and TypeSpec define operations and data structures holistically, they enable the extraction of rich dependency graphs directly from their OpenAPI or AST representations. This capability allows the discovery of logical relationships between endpoints based solely on their structural definitions without the need for runtime execution or manual inspection.

The resulting analyses achieved great precision: the generated dependency graphs were noise-free and correctly identified object-level dependencies, effectively mapping the semantic flow between endpoints. In comparative evaluations, particularly with the Smithy and TypeSpec models, the results surpassed those produced by traditional static and dynamic analysis methods. The graphs not only revealed more meaningful relationships but also provided a higher-level view of system interactions, capturing dependencies between entire objects rather than isolated primitive fields.

However, several limitations were identified. The gRPC model lacks the ability to define wrapped primitives. This absence leads to information loss in the generated OpenAPI specification and, consequently, to reduced accuracy in dependency detection. Furthermore, the quality of the results produced by our analysis proved to be tightly coupled with the quality of the model definition itself. The introduction of well-structured wrappers and consistent type hierarchies greatly enhances dependency resolution accuracy, whereas loosely modeled or inconsistent

schemas can lead to missed or ambiguous connections. Simply put, the better the model structure, the clearer and more meaningful the resulting dependency graph.

Looking ahead, a promising direction for future research involves exploring how Smithy and TypeSpec can be utilized to model and implement Model Context Protocol (MCP) servers following a Model-First design approach. The Model Context Protocol, recently introduced by Anthropic, defines a standardized mechanism for enabling AI systems and tools to exchange structured context and interact programmatically through well-defined interfaces. Since both Smithy and TypeSpec are capable of precisely describing operations, input/output schemas, and service behaviors, they could serve as ideal foundational languages for formally specifying MCP tools and services.

Bibliography

[1] "SmartBear, State of Software Quality 2023, 2023," [Online]. Available: https://smartbear.com/state-of-software-quality/api/.

[2] M. Fowler, "Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2021," [Online]. Available: https://martinfowler.com/books/eaa.html.

[3] "Amazon Web Services, AWS Smithy: A Language for Defining Services and SDKs," [Online]. Available: https://smithy.io.

[4] "TypeSpec Language: Model-First API Design and Code Generation," [Online]. Available: https://typespec.io.

[5] "Protocol Buffers and gRPC Overview," [Online]. Available: https://grpc.io/docs/.

[6] N. Ford, R. Parsons, and P. Kua, "Building Evolutionary Architectures: Support Constant Change, 2nd ed., O'Reilly Media, 2022," [Online]. Available: https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532/.

[7] Π. Παναγιώτης, "Αυτόματη παραγωγή τεκμηρίωσης εξαρτήσεων μεταξύ κλήσεων endpoints ενός REST API," [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/18728.

[8] Δ.-Δ. Γεροκωνσταντής, "Dynamic Analysis of Inter-endpoint Dependencies in RESTful APIs," [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19419.

[9] Panagiotis Papadeas, Dimitrios Gerokonstantis, Christos Hadjichristofi and Vassilios Vescoukis, "Enhancing API documentation by inter-endpoint dependency graphs," 2025. [Online]. Available: http://dx.doi.org/10.15439/2025F8035.

[10] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[11] "OASIS OpenAPI Initiative, OpenAPI Specification," [Online]. Available: https://swagger.io/specification.

[12] "Swagger UI," [Online]. Available: https://swagger.io/tools/swagger-ui.

[13] "Swagger Codegen," [Online]. Available: https://swagger.io/tools/swagger-codegen.

[14] "Stoplight," [Online]. Available: https://stoplight.io.

[15] "gRPC Metadata," [Online]. Available: https://grpc.io/docs/guides/metadata.

[16] "gRPC Interceptors," [Online]. Available: https://grpc.io/docs/guides/interceptors.

[17] "gRPC Health Checking Service," [Online]. Available: https://grpc.io/docs/guides/health-checking/.

[18] "Grpc Gateway," [Online]. Available: https://github.com/grpc-ecosystem/grpc-gateway.

[19] "Postman gRPC interface," [Online]. Available: https://learning.postman.com/docs/sending-requests/grpc/grpc-request-interface/.

[20] "gRPC Web," [Online]. Available: https://github.com/grpc/grpc-web.

[21] "Bridging the Gap Between REST and gRPC," [Online]. Available: https://zuplo.com/learning-center/grpc-api-gateway.

[22] "Introducing AWS API models and publicly available resources for AWS API definitions," [Online]. Available: https://aws.amazon.com/blogs/aws/introducing-aws-api-models-and-publicly-available-resources-for-aws-api-definitions/.

[23] "AWS Smithy: The Next Level of API Modeling," [Online]. Available: https://aws.plainenglish.io/aws-smithy-the-next-level-of-api-modeling-f144c0b6c422.

[24] G. Hohpe, "Cloud strategy and design patterns: Architecting scalable systems.," [Online].