ISSN: 3065-0003 | Volume 2, Issue 4

Open Access | PP: 81-86

DOI: https://doi.org/10.70315/uloap.ulirs.2025.0204014

Approaches to Automating Architectural Structure Generation and Integrity Verification

Dmitrii Kuzmin

Senior Software Engineer, EPAM Systems (Georgia) LLC, Batumi, Georgia.

Abstract

The study focuses on engineering techniques that automate the generation of architectural structures and systematic verification of their integrity in large front-end codebases for state-regulated digital services in education and healthcare. The work combines an analytical review of recent research on evolutionary software architecture, static analysis, architecture conformance checking, code generation, and accessibility testing with a detailed examination of an industrial case involving Moscow Electronic School (MES) and Electronic Medical Records (EMR). Template-based generators (PlopJS, Angular Schematics, Nx generators) are examined as a way to encode architectural decisions and eliminate manual boilerplate. Static-analysis–driven "fitness functions" implemented through ESLint, custom plugins, and CI hooks are analysed as an automated guardrail for accessibility, dependency boundaries, and architectural coupling. The article formulates an integrated approach where generators construct a consistent architecture skeleton, while fitness-function checks prevent erosion during everyday development. The research generalizes engineering experience into a reusable scheme for front-end projects that operate under strict governmental requirements for reliability, traceability, and accessibility. It outlines directions for the quantitative evaluation of time savings and defect reduction in future work.

Keywords: *Software Architecture Automation, Code Generation, Angular Schematics, Nx Generators, ESLint, Fitness Functions, Accessibility, Architecture Conformance Checking, Static Analysis, Monorepo.*

INTRODUCTION

Digital platforms in public education and healthcare combine rich interactive interfaces, complex domain logic, and strict regulatory constraints. Architectures of such systems tend to evolve around monorepos, shared UI libraries, state management layers, and numerous domain-specific features. Manual creation of architectural artifacts in these environments leads to inconsistent structures, accidental coupling, and degradation of non-functional qualities, such as accessibility and maintainability. At the same time, research on evolutionary architecture and static analysis demonstrates that the structural properties of systems can be treated as measurable objectives and verified automatically throughout their life cycle.

The article addresses this tension between architectural intent and daily development practices. The primary goal is to describe and generalize approaches that reduce manual effort in building and evolving front-end architecture while preserving integrity under continuous change in government-grade systems, such as learning management environments and Electronic Medical Records.

The research agenda includes three closely related tasks:

1) To analyse contemporary work on static-analysis-based architecture conformance, code quality metrics, fitness-function ideas, and accessibility checking to identify techniques suitable for front-end projects built with JavaScript and TypeScript;
2) to systematise practical patterns of template-based generation of architectural structures using PlopJS, Angular Schematics, and Nx generators, including their application in real projects where a single feature requires coordinated changes in two dozen files;
3) to formulate an integrated scheme that combines generators with fitness-function checks implemented through ESLint, Husky pre-commit hooks, and CI pipelines, with special attention to accessibility and import-boundary rules that are critical for public-sector interfaces.

Scientific novelty lies in bridging theoretical work on evolutionary architecture and architecture conformance with concrete tooling for front-end development. The article treats template generators and lint rules as an executable architectural specification tuned for state-regulated digital services. It reconstructs such a specification based on an industrial case from Moscow Electronic School (MES) and Electronic Medical Records (EMR).

MATERIALS AND METHODS

The study relies on a corpus of recent research and practice-oriented publications that deal with automated architecture governance, static analysis, and code generation. A.S. Abdelfattah [1] investigates the evolution of microservice systems using static analysis, introducing metrics that capture changes in inter-service communication and data dependencies across system versions. O. Carter [2] surveys code-quality metrics and static-analysis tools, such as ESLint, SonarQube, and Pylint, with an emphasis on their integration into CI pipelines and their impact on engineering practices. N. Chondamrongkul [3] studies software evolutionary architecture and automated planning for functional changes, where search-based techniques rely on objective functions close in spirit to architectural fitness functions. N. Kodali [4] analyses Angular CLI and Schematics as a mechanism for automatic generation and modification of Angular artifacts and quantifies the effect of such tooling on development time and structural consistency. L. Maharjan [5] evaluates the use of GPT-4o for resolving accessibility issues reported by eslint-plugin-jsx-a11y and similar static checkers, showing how language models can complement rule-based analysis of accessibility defects. R. Opdebeeck [6] develops a static-analysis framework for Ansible infrastructure-as-code artifacts, using graph-based intermediate representations (Joern) and architecture conformance metrics for cloud deployments. M. Resende [7] proposes ArchLintor, a static-analysis tool that detects violations of architectural rules in TypeScript systems by encoding intended dependencies between modules and examining source code. D.G. San Martín Santibáñez [8] presents REMEDY, an architecture conformance approach for self-adaptive systems, where a domain-specific language describes planned architecture and a static analysis recovers current architecture for comparison. S.L.J. Shabu [9] describes the development of an e-commerce system using a MEAN stack and Nx monorepo, highlighting the benefits of generators and workspace tooling for enforcing consistent project structure. M. Zhong [10] introduces ScreenAudit, an LLM-based accessibility analyser for mobile applications that builds on existing rule-based checkers and demonstrates significantly wider coverage of screen-reader issues.

Methodologically, the work follows an analytical and comparative approach. First, publications [1–10] are examined to identify the canonical building blocks for automated architecture governance, including intermediate representations, rule engines, template-based generators, and CI integration patterns. Second, engineering practices in the MES, EMR, and 12 STOREEZ warehouse management system (WMS) projects are reconstructed, with detailed mapping of generator templates (PlopJS in EMR, Angular Schematics and Nx generators in 12 STOREEZ), lint rules, and pre-commit hooks, all tied to architectural decisions. Time savings are estimated using expert judgement based on repeated use of generators for routine components and complex “card” features. Third, the findings are generalized into a layered scheme that links structural templates with fitness-function checks, forming a blueprint for front-end architecture automation in public-sector systems.

RESULTS

Analysis of the literature confirms that architectural integrity can be formalised through metrics and rule sets derived from static representations of software artifacts. Abdelfattah et al. introduce a method for reconstructing service and data views of microservice systems from source code, then use these views to measure architectural evolution and detect erosion of coupling patterns over time [1]. Static analysis extracts endpoint declarations and call sites, which are matched to produce a dependency matrix between services and between services and data entities. Similar graph-based representations are also found in Opdebeeck’s work on infrastructure-as-code, where Ansible playbooks are transformed into code property graphs to detect code smells and non-compliance with best practices [6]. These studies show that once a system is mapped into a graph or matrix, integrity checks can be reduced to automated queries expressed as metrics or patterns.

Evolutionary-architecture research further strengthens this view. Chondamrongkul and Sun propose automated planning for functional changes based on search algorithms that optimise architectural objectives [3]. Instead of treating architecture as a fixed blueprint, their method evaluates change plans using objective functions tied to qualities such as coupling or latency. Although their focus lies on service-level refactorings, the underlying idea is closely related to fitness functions: architectural properties become measurable targets used to drive or veto changes. This theoretical foundation supports the concept of encoding front-end architectural constraints in rule engines, where each rule corresponds to a fitness function that continuously evaluates the health of the structure.

The static-analysis tools discussed by Carter illustrate how such fitness functions are operationalized in practice [2]. He surveys tools like ESLint, SonarQube, and Pylint in the context of CI/CD pipelines. He emphasises that rule sets can be tuned to reflect architectural and organisational conventions, not only low-level code smells. In Opdebeeck’s thesis, similar principles are applied to infrastructure scripts: architecture-conformance metrics and security policies are translated into static analyses executed over a graph representation of deployment code [6]. San Martín Santibáñez et al. take a step further in REMEDY by introducing a domain-specific language to describe the planned architecture for self-adaptive systems and a conformance process that compares this specification with a recovered architecture model [8]. In combination, these works suggest a pattern where architectural integrity is enforced by (1) a declarative description of intended structure and (2) automated analysis that flags deviations.

On the front-end side, Kodali analyses Angular CLI and Schematics as a means of encoding recurring architectural patterns [4]. Schematics represent instructions for creating and modifying files, including components, services, modules, and configuration artifacts. The empirical evaluation in that article suggests that standard generators reduce development time for new features and enhance consistency across projects. Shabu and co-authors report similar conclusions for a MEAN-stack e-commerce system built on an Nx monorepo: generators streamline the creation of modules, services, and APIs, while the monorepo tooling maintains a consistent dependency graph between applications and libraries [9]. Nx generators are described as TypeScript functions that scaffold whole slices of functionality and insert them into existing routing and configuration files.

In the EMR and 12 STOREEZ projects, template-based generation follows the same philosophy. Still, it reaches deeper into the architecture, while MES primarily relies on ESLint-based static checks without custom code generators. A generator for a complex “card” feature in an EMR system creates approximately fifteen new files across several layers (connectors, model, UI composition, and routing). It modifies roughly seven existing files responsible for document name dictionaries, card registry indices, counters, and application bootstrap. Manual execution of this sequence previously demanded knowledge of more than twenty integration points. In practice, a generator call for a simple component, model, or modal window saves approximately three to ten minutes of routine work and eliminates mechanical errors in naming or imports. For a new card feature, the saving grows to twenty to forty minutes for an experienced engineer and over an hour for a newcomer who would otherwise search for all relevant files and correct inevitable mistakes. The generator not only creates code but codifies architectural decisions, including adherence to a shared UI kit, uniform state management patterns, and feature-sliced file placement.

Figure 1 synthesises these observations by presenting a conceptual curve that relates the degree of automation coverage to adequate development time per feature. The shape follows the hypothetical chart published by Kodali, where enabling Angular productivity features such as schematics, code generation, and workspace tooling progressively reduces implementation time for new modules [4]. Here, the curve is reinterpreted for a broader ecosystem that includes PlopJS generators for pages and cards in the EMR system, Nx generators for workspace maintenance in the 12 STOREEZ monorepo, and scripted insertion of imports and routes for medical-records modules.

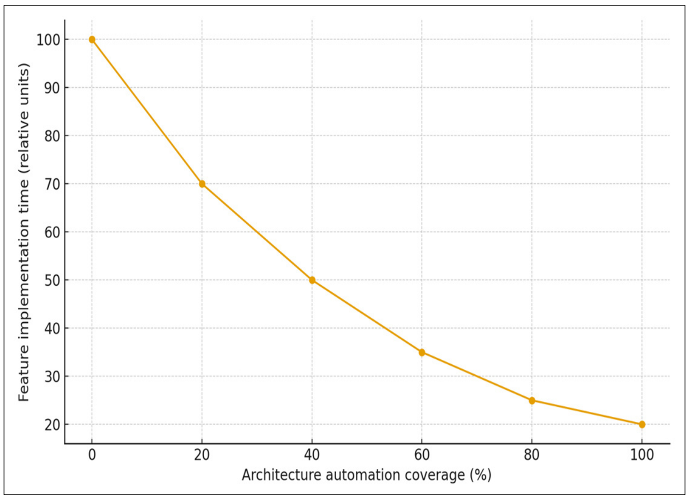

Figure 1. Conceptual relationship between architecture automation coverage and feature implementation time (compiled by author based on data layout from [4])

Automation of the structure alone does not guarantee integrity. The second pillar in the proposed approach consists of fitness-function checks implemented with static-analysis tools. Resende’s ArchLintor for TypeScript exemplifies how architectural rules can be encoded in a linter: intended dependencies between modules are specified, and the tool reports every import that contradicts these rules [7]. REMEDY, with its DSL for planned architectures and conformance engine, illustrates a similar separation between specification and implementation for self-adaptive systems [8]. In front-end monorepos, ESLint and its ecosystem of plugins form a natural platform for such rules. Existing plugins already cover style conventions, complexity limits, and many accessibility requirements; extensions such as eslint-plugin-boundaries introduce rule sets that restrict imports between layers and packages by declarative configuration.

Accessibility-related fitness functions represent a separate class of checks. Maharjan analyzes GPT-4 prompts aimed at resolving issues reported by eslint-plugin-jsx-a11y and shows that static checkers reliably flag recurring problems but often produce terse messages. At the same time, language models help propose concrete corrections [5]. Zhong’s ScreenAudit extends the idea further: it combines traversal of app screens, metadata extraction, and LLM-based reasoning to uncover screen-reader issues that rule-based tools miss, achieving coverage more than twice that of a widely used accessibility checker [10]. In government interfaces, where accessibility requirements are non-negotiable, such findings justify a strict regime: any a11y violation detected by ESLint or related tools can be configured as an error, breaking the build in CI rather than remaining a warning.

Experience from MES and EMR demonstrates how these ingredients work together in daily practice. Architecture rules capture feature-sliced boundaries: UI libraries do not import application code, pages depend only on core and shared layers, and cross-feature imports are permitted only through stable interfaces. ESLint with a boundaries plugin enforces these rules, while additional custom rules guard against direct access to low-level API modules or bypassing state-management abstractions. Accessibility plugins examine JSX or template code for missing ARIA attributes, insufficient contrast, and improper focus management. Husky pre-commit hooks run the linter and selected tests locally; CI pipelines repeat the checks for every merge request. When a generator introduces a new feature, its output already respects these constraints. Hence, any divergence arises only when developers attempt to modify generated code in ways that contradict the architectural specification.

From a theoretical viewpoint, the combined mechanism mirrors the architecture-conformance frameworks described in [1; 6–8]: there is a planned structure expressed implicitly in templates and lint configuration, and a current structure that surfaces in the actual codebase. Static analysis operates on the latter to detect discrepancies. The difference lies in the engineering emphasis on front-end tooling and the granularity of artifacts: instead of microservices or infrastructure scripts, the unit of analysis becomes a feature, a card, a component, and the imports connecting them. Nevertheless, metrics such as the number of cross-layer dependencies, density of forbidden imports, or proportion of components passing a11y checks can be interpreted analogously to microservice coupling and IaC smell counts.

In my view, the primary practical outcome is the emergence of an executable architectural specification tailored to front-end applications in state-regulated domains. Generators encode how architects expect features to be structured; fitness functions encoded in linters and CI encode how this structure should evolve under change. Together, they reduce reliance on oral tradition and scattered documentation, aligning with the broader trends in evolutionary architecture and automated governance observed in the literature.

DISCUSSION

The reviewed studies on static analysis and architecture conformance cover different domains but share a familiar pattern: they construct some intermediate representation and run rule-based analyses over it. Abdelfattah et al. reconstruct service and data views of microservice systems [1]. Opdebeeck uses code property graphs for infrastructure scripts [6]. Resende encodes intended dependencies between TypeScript modules [7], while San Martín Santibáñez describes a planned architecture using a DSL for self-adaptive systems [8]. Table 1 summarizes these approaches to the analyzed artifacts, representation, and explicit architecture rules.

Table 1. Static-analysis approaches to architecture and quality governance based on sources [1; 6–8]

Source	Analyzed artifact	Representation	Focus of rules	Relevance for front-end monorepos
Abdelfattah et al. [1]	Microservice implementations	Service and data dependency matrices	Evolution of coupling and cohesion across versions	Inspires measurement of dependency evolution between front-end features and shared libraries
Opdebeeck [6]	Ansible infrastructure-as-code	Code property graphs and smell detectors	Security and maintainability of deployment scripts	Suggests the use of graph-based analysis for building and CI configuration in monorepos
Resende et al. [7]	TypeScript source code	Architecture conformance rules over imports	Violations of intended layering and module boundaries	Directly applicable to ESLint-based boundary rules in TypeScript front-end projects
San Martín Santibáñez et al. [8]	Self-adaptive systems	DSL specification versus recovered architecture	Deviations from reference model (MAPE-K)	Provides a model for explicit architectural DSL in front-end projects, e.g., for feature-sliced constraints

Textual evidence in these works supports the conclusion that once an architectural view is extracted, declarative rule sets become the main instrument for enforcing integrity. For front-end codebases, imports between modules, directories, and packages already form a graph similar to service or IaC graphs. This encourages reuse of the same pattern: a generator defines intended edges; a linter checks actual edges, and metrics based on these edges provide feedback on structural erosion.

A parallel line concerns code-generation and workspace automation. Chondamrongkul's evolutionary-planning work illustrates how automation can explore large search spaces of possible architectural changes [3]. Kodali's study of Angular CLI and Schematics describes a more developer-centric scenario where predefined schematics instantiate patterns for components, modules, and configurations, leading to reduced implementation time and more homogeneous codebases [4]. Shabu's MEAN-stack e-commerce system demonstrates the effect of combining the Nx monorepo tooling with such generators to maintain consistency across multiple applications and services [9]. Carter's overview of static-analysis tooling emphasises that generators and linters reach their full potential only when integrated into CI/CD workflows where each commit passes through code-quality gates [2]. Zhong's ScreenAudit complements this picture by showing that a11y testing benefits from multilayered automation: static rules, runtime checks, and LLM-based reasoning jointly increase coverage [10].

Table 2 consolidates these observations for automation techniques that influence architectural structure and its verification.

Table 2. Automation techniques influencing architectural structure and integrity verification based on sources [2–5; 9–10].

Source	Automation mechanism	Architectural impact	Integration pattern
Carter [2]	Static-analysis tools and CI quality gates	Codifies coding standards, basic structural constraints, and test coverage thresholds	Integrated into CI; blocks merge when checks fail
Chondamrongkul [3]	Search-based planning for architecture evolution	Evaluates change plans against structural objectives such as coupling	Used during design and refactoring rather than per-commit
Kodali [4]	Angular Schematics and CLI generators	Standardises the structure of components, modules, and configuration	Invoked by developers to scaffold new features and refactor existing ones
Maharjan [5]	LLM-based repair of a11y issues from eslint-plugin-jsx-a11y	Enhances accessibility while preserving the structure suggested by static rules	Runs as a complement to linting, not as a gatekeeper
Shabu et al. [9]	Nx monorepo generators and workspace tooling	Maintains coherent structure across multiple applications and libraries	Combined with a monorepo dependency graph and shared configuration
Zhong et al. [10]	ScreenAudit combines LLM and metadata extraction	Extends detection of accessibility defects beyond static rules	Evaluated periodically to audit releases

Industrial practices in MES, EMR, and 12 STOREEZ align well with the patterns described in Table 2. In EMR, PlopJS generators act as front-line tools that encode architectural blueprints for features, cards, and integration points, while in 12 STOREEZ, the same function is performed by Angular Schematics and Nx generators. Across all three systems, ESLint, boundary rules, and a11y plugins serve as continuous quality gates similar to those described by Carter [2], with Husky and CI pipelines enforcing their execution on each commit and merge. From the broader literature's perspective, this configuration converges on a layered governance system: generators and monorepo tooling define the structure; static analysis validates it; periodic audits, and potentially LLM-based tools like those studied by Maharjan and Zhong, refine code where rules are insufficient [5; 10].

In my opinion, one of the most significant outcomes of such a configuration is a reduction in architectural drift caused by everyday changes. Without generators, each new card in a medical-records system requires developers to remember dozens of structural and integration steps; without stringent linting, nothing prevents a well-meaning engineer from bypassing established boundaries for the sake of a quick fix. With the combined setup, a developer launches a generator that produces a feature skeleton aligned with architectural intent, then relies on automated checks to catch any deviation introduced during manual customisation. This structure resembles REMEDY's planned-versus-current-architecture cycle [8], but adapted for front-end workstations and TypeScript tooling.

The main limitation of the present study stems from its analytical nature. The literature provides quantitative results on microservice evolution, IaC smells and accessibility coverage [1; 6; 10], while the MES/EMR case offers only expert estimates for time savings and qualitative observations about reduced cognitive load. A more rigorous evaluation would require logging generator invocations, measuring manual effort saved, and tracking the number of architecture violations prevented by ESLint rules over a substantial period. Such a study would connect the conceptual framework outlined here with empirical evidence comparable to that reported for microservices and infrastructure.

CONCLUSION

The analysis shows that architectural structure in large front-end systems can be encoded explicitly in two complementary forms: generator templates that produce consistent sets of files and connections, and fitness-function rules that inspect code for structural and accessibility violations. Together, these elements form an executable description of the intended architecture for state-regulated digital services.

The review of recent research on static analysis, architecture conformance, and evolutionary planning confirms that graph-based representations and declarative rules provide a robust foundation for such automation. At the same time, studies on accessibility checking and LLM-assisted repair indicate that fitness functions for a11y require both rule-based and semantic reasoning.

The industrial cases from MES, EMR, and 12 STOREEZ illustrate how this framework materialises in practice. In EMR, PlopJS generators reduce the cost of introducing new features by automating the creation of dozens of files and their integration into routing, state management, and UI layers, while in the 12 STOREEZ monorepo the same effect is achieved by Angular Schematics and Nx generators. Across all three systems, ESLint rules, accessibility plugins, and pre-commit hooks prevent erosion of feature-sliced boundaries and block the introduction of non-compliant interfaces into production. The three tasks formulated in the introduction are addressed through the synthesis of literature, reconstruction of engineering practices, and formulation of a unified scheme for automated architectural structure generation and integrity verification.

This scheme suits not only governmental educational and medical systems but also any long-lived software system where architecture must remain predictable under continuous change. Further research can extend it with explicit architectural DSLs, richer metrics for front-end coupling, and quantitative analysis of the trade-offs between automation effort and maintenance gains.

REFERENCES

1. Abdelfattah, A. S., Cerny, T., Yero Salazar, J., Li, X., Taibi, D., & Song, E. (2024). Assessing evolution of microservices using static analysis. Applied Sciences, 14(22), 10725. https://doi.org/10.3390/app142210725
2. Carter, O. D. (2024). Advancing software quality: A comprehensive exploration of code quality metrics, static analysis tools, and best practices. Journal of Science & Technology, 5(1), 69–81. URL: https://thesciencebrigade.com/jst/article/view/62
3. Chondamrongkul, N., & Sun, J. (2023). Software evolutionary architecture: Automated planning for functional changes. Science of Computer Programming, 230, 102978. https://doi.org/10.1016/j.scico.2023.102978
4. Kodali, N. (2024). The evolution of Angular CLI and Schematics: Enhancing developer productivity in modern web applications. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 10(5), 805–812. https://doi.org/10.32628/CSEIT2410428
5. Maharjan, L. (2025). A study on the capabilities of GPT-4o for accessibility-related issues (Master's thesis). Miami University. URL: https://etd.ohiolink.edu/acprod/odb_etd/ws/send_file/send?accession=miami1753447762551235&disposition=inline
6. Opdebeeck, R. (2024). Static analysis for quality assurance of Ansible infrastructure-as-code artefacts (Doctoral dissertation). Vrije Universiteit Brussel. URL: https://soft.vub.ac.be/Publications/2024/vub-soft-phd-25102024-rubenopdebeeck.pdf
7. Resende, M., Durelli, R. S., & Terra, R. (2025). ArchLintor: Detecting architectural violations in TypeScript systems. In Proceedings of the XXXIX Brazilian Symposium on Software Engineering (SBES 2025) (pp. 900–906). https://doi.org/10.5753/sbes.2025.11075
8. San Martín Santibáñez, D. G., Camargo, V., & Angulo, G. (2025). Architecture conformance checking for self-adaptive systems. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.5174803
9. Shabu, S. L. J., Kumar, S. P., Pranav, R., Refonaa, S., & Maheswari, M. (2023). Development of an e-commerce system using MEAN stack with NX monorepo. In Proceedings of the 2023 7th International Conference on Trends in Electronics and Informatics (ICOEI). https://doi.org/10.1109/ICOEI56765.2023.10125948
10. Zhong, M., Chen, R., Chen, X., Fogarty, J., & Wobbrock, J. O. (2025). ScreenAudit: Detecting screen reader accessibility errors in mobile apps using large language models. In CHI '25: Proceedings of the CHI Conference on Human Factors in Computing Systems, 1163, 1–19. https://doi.org/10.1145/3706598.3713797