Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 1

Open Access | PP: 116-121

DOI: https://doi.org/10.70315/uloap.ulete.2026.0301020

Rendering Interactive Generative User Interface Components Based on React Libraries When Working with Large Language Models

Elias Tounzal

Software Engineer at Quaestor Technologies, Inc., USA.

Abstract

The article examines the transition from dialog assistants to the generation of interactive user interfaces powered by React libraries and large language models, driven by increasing expectations for turnkey operation and by the limitations of linear chat for complex, multi-step tasks. The purpose of the study is to conceptualize and compare engineering trajectories for generating React interfaces from LLM output, and to formulate architectural principles that ensure behavioral predictability, controllable interactivity, and robust user experience. The relevance of the research is determined by the rapid adoption of generative systems in analytical dashboards, administrative interfaces, and visual analytics scenarios, where iterative query refinement is required without loss of context and with preserved control over data and security. The novelty of the work lies in the systematic comparison of the approach of directly generating component source code with that of generating a structured specification, followed by validation and rendering a tree of pre-selected React components. It is shown that the specification track, in combination with caching, layout contracts, partial updates, and mature testing and observability practices, enables reducing the gap between formal correctness and functional suitability of generated interfaces and localizing risks at the boundary between natural language and executable behavior. The article is intended for researchers of dialog and interface systems, frontend architecture engineers, design system developers, and practitioners implementing generative UI pipelines in products with high interactive load.

Keywords: *Generative User Interfaces, Large Language Models, React, Interface Specification, Component Registry.*

INTRODUCTION

The transition from dialog assistants to interface generation has been shaped by two parallel developments. Dialog systems now sustain context and support multi-step tasks, which has increased expectations for complete task execution rather than simple reference answers. At the same time, linear chat limits efficient navigation through complex information and reduces user control during iterative refinement and verification (Singh & Beniwal, 2021; Ross et al., 2023).

In this study, interactive generative user interface rendering is defined as a process in which a large language model produces a specification of interface structure and behavior that is transformed into a React component tree and displayed with preserved interactivity. The generated result is a controllable composition of visual elements linked to state and data. Interactivity is realized through events and actions that can update state, request data, and initiate a new generation cycle (Wang et al., 2022).

Typical applications include dashboards, administrative panels, form generation, and interactive analytics, where users submit requests in natural language and receive visualizations, filters, and controls for further refinement. This iterative logic is especially important in analytics and has also been demonstrated in systems for visualization through continuous natural language input (Tabalba et al., 2022).

MATERIALS AND METHODOLOGY

The study draws on a corpus of 11 sources devoted to rendering interactive generative UI components with React in large language model workflows. These materials cover the evolution of dialog agents and behavioral predictability,

Citation: Elias Tounzal, "Rendering Interactive Generative User Interface Components Based on React Libraries When Working with Large Language Models", Universal Library of Engineering Technology, 2026; 3(1): 116-121. DOI: https://doi.org/10.70315/uloap.ulete.2026.0301020.

conversational assistants for software development, language models as intermediaries between user intention and interface operations, and natural-language-driven visualization systems with iterative editing (Singh & Beniwal, 2021; Ross et al., 2023; Wang et al., 2022; Tabalba et al., 2022). The engineering perspective is supported by work on DSL-based interface specification and model-driven UI development, alongside studies on design systems, code generation quality, reuse of verified fragments, and dependency-management risks in software frameworks (Chavarriaga et al., 2023; ErazoGarzón et al., 2023; Lamine & Cheng, 2022; Ni et al., 2024; Gou et al., 2024; Tan et al., 2026).

Methodologically, the research applies qualitative comparative analysis to two engineering trajectories for deriving a React interface from LLM output. The first examines direct component code generation in light of observed variability in language-to-code tasks. The second considers the generation of a structured specification followed by schema-based validation and transformation into a tree of pre-selected components, which is assessed through the principles of DSLs and model-driven UI design (Ni et al., 2024; Chavarriaga et al., 2023; ErazoGarzón et al., 2023). The analysis also considers interactivity stabilization through component registries and design systems, triangulates architectural reasoning with studies of iterative visual analytics and controllable work artifacts, and evaluates execution risks related to dependencies and software supply-chain security (Lamine & Cheng, 2022; Tabalba et al., 2022; Ross et al., 2023; Tan et al., 2026).

RESULTS AND DISCUSSION

In the context of interface generation based on React libraries, two stable approaches are used. The first approach is organized around generating component source code, where the model outputs ready-made fragments of presentation and associated logic intended for assembly into an application. This path is attractive due to its high expressiveness, since the model can describe virtually any combination of components and states within the available dependencies. At the same time, it is vulnerable to syntactic failures, mismatches with the project's stylistic and architectural constraints, and the emergence of latent side effects that are difficult to track during automated assembly. Tasks of transforming natural language into program text are characterized by sensitivity to formulation, context, and test coverage, which makes result quality heterogeneous and weakly predictable when transferred across domains. These characteristics are clearly evident in studies evaluating program text generation from descriptions, where the divergence between formal correctness and functional suitability across diverse task formulations is highlighted (Ni et al., 2024).

The second approach relies on generating an interface specification as a structured description that enumerates admissible elements, their properties, connections to data, and rules for reacting to events. The specification then undergoes schema validation and is transformed into a tree of pre-selected components, which enhances controllability and simplifies security control, since the set of allowed constructs is fixed prior to invoking the model. Conceptually, this approach is consistent with the engineering tradition of domain-specific description languages, in which the level of abstraction is raised toward the task, thereby eliminating a portion of errors arising from direct low-level code writing (Chavarriaga et al., 2023). For practical systems, a hybrid construction often proves productive, in which the main trajectory is specified by the specification track, while targeted extensions are allowed through strictly constrained insertions accompanied by additional validation and sandboxed execution. The logic of hybridity is closely related to the idea of combining generation with the retrieval and reuse of vetted fragments, which, in program synthesis tasks, reduces the risk of quality degradation for rare libraries and atypical requirements (Gou et al., 2024).

The reference pipeline architecture is based on a sequence of transformations in which the user's initial input is converted into a formalized intention, then bound to the application's data and state, and, after that, an interface plan is formed as either a specification or program text. At the next step, the plan is validated against the schema and the component security policy; then the renderer constructs a tree of components from the React library and launches an event cycle in which user actions trigger state updates and data requests, and, when necessary, initiate rebuilding the plan. This organization aligns with practices in interactive visual analytics, where stable latency management and the reproducibility of interaction sequences are central. These issues are formalized by modeling the interaction space and analyzing user action logs, enabling the designer to relate intentions, latencies, and interface changes.

Within this architecture, truth is stored in three interrelated layers. The first layer is the interface state, which includes selected filters, form parameters, and the current task context. The second layer is data and their derived representations, which must be updated deterministically and kept separate from the generative mechanism. The third layer is the component policy, which includes the registry of admissible elements and constraints on properties and events, since it defines the boundaries of the interface plan produced by the model. Caching is located at the boundaries between intention and data, as well as between data and rendering, so that repeated queries and transformation steps yield identical results at reduced cost. In interactive systems where users expect responses within hundreds of milliseconds, this is consistent with research on real-time query support and latency management in exploratory analysis. The reference pipeline architecture is shown in Figure 1.

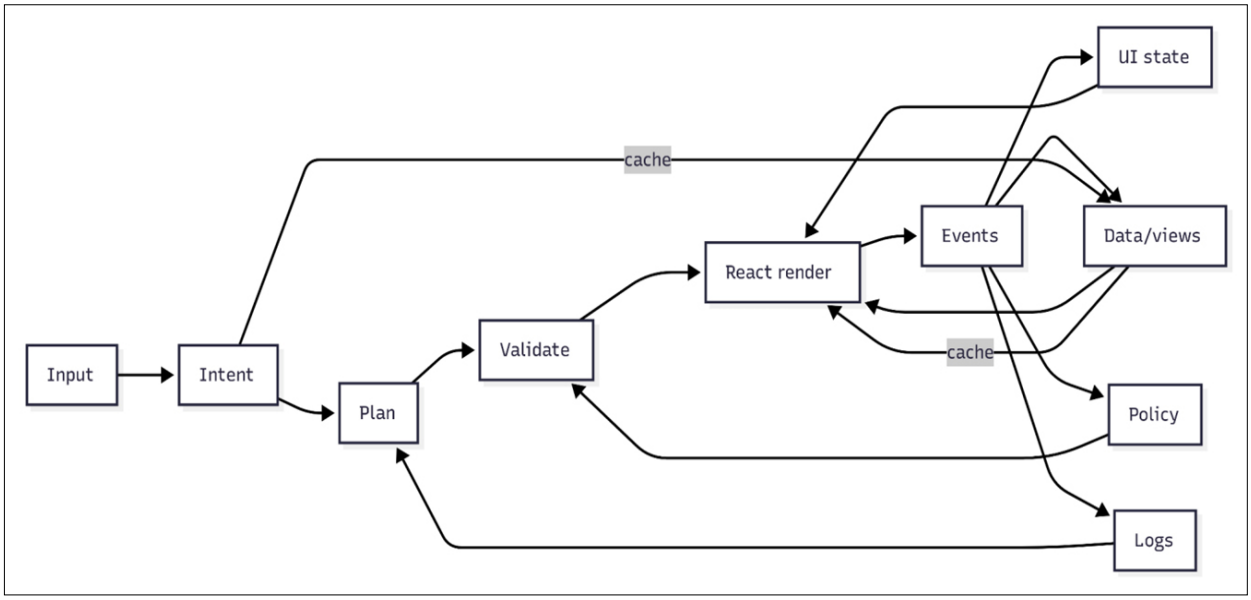

Fig. 1. Reference pipeline architecture

The component registry defines a practical boundary between the generative mechanism and the runtime environment. It thus logically continues the policy stage of the pipeline and fixes the admissible solution space for the interface plan. The core principle of such a registry can be formalized as an allowlist. The model is permitted to assemble the interface only from pre-defined building blocks and only with properties that have undergone typing and safety verification. This reduces the likelihood that the component tree will contain constructs violating data access rules, accessibility requirements, or performance constraints. From the standpoint of design system practice, the registry serves as the locus where component values and behavior are stabilized, and changes are propagated in a controlled manner, with regard for cross-product compatibility, as emphasized in an empirical study of design system maintenance and evolution in development teams (Lamine & Cheng, 2022).

It is reasonable to group registry content by functional categories, since this simplifies both generation and subsequent plan validation. Layout components define grids, spacing, and containers that ensure predictable structure and stable keys for differential updates. Form components include input, selection, validation, and error messages, because forms constitute the primary point of contact between user intention and data (Xiao et al., 2024). Table and chart components serve data representation and exploration, where consistency of filters, sorting, and linkages among views is required. Modal and notification components serve transient contexts, confirmations, and operational feedback, reducing cognitive load and supporting the event-driven interaction cycle. This decomposition is consistent with approaches in model-driven interface development, where specifications are used for systematic design and generation, and the library of visual elements is treated as a managed vocabulary of representations (ErazoGarzón et al., 2023).

Registry dependencies must be managed with the same rigor as the components themselves, since the generative interface amplifies the risks of transitive dependencies and incompatible updates. At the implementation level, this implies an explicit description of each registry element's dependencies and compatibility checks with the design system, including design tokens and theming rules, so that the model does not lead to drift in the visual language across screens and products. At the security and maintenance level, this implies control over software component supply chains, as a dependency may be functionally correct while simultaneously creating a latent attack surface. Research on dependency management and patching shows that a substantial share of changes in ecosystems is driven by optimization and compatibility, and that explicit attention to security occurs less frequently than would be expected from stated priorities (Tan et al., 2026). This strengthens the argument for the registry as a point of enforced control where updates pass through an admission policy, testing, and reversible migration. The Registry Governance Summary Matrix is shown in Table 1.

Table 1. Registry Governance Summary Matrix

Block	What it is	What the registry must lock down
Purpose	Boundary generation - runtime, extends the policy stage	Component allowlist
Rule	Only pre-defined building blocks + verified props	Prop typing/schema + safety constraints
Categories	Make generation + validation easier	layout · forms · data (tables/charts) · overlays (modals/notifications)

Layout	Grid/containers/spacing, stable structure	Spacing tokens + stable keys
Forms	Input/selection/validation/errors	Field contracts + error rules
Data views	Filters/sorting/linked views	Unified filters/sorting API
Overlays	Confirmations + operation feedback	Event/status types + timing rules
Dependencies	Transitive + breaking-change risk	Explicit deps + compatibility/theming checks
Updates control	Admission, tests, reversible migration	Versioning + deprecations + rollback

State and data management in a generative interface requires separating two sources of variability. Interface state describes user choices, open panels, local settings, and intermediate input values. Data state describes information obtained over the network, including its freshness, provenance, and reuse rules. This separation is necessary because the same user intention may lead to different data requests, and the same data may underlie different visual representations depending on context. As a result, the interface state layer must be compact and resilient to restructuring of the component tree. The data state layer must be oriented toward reconciliation with the network and caching. The generative mechanism is connected to these layers and accesses them via a constrained protocol that expresses reads and writes as actions rather than as direct side effects.

Interactive input and network requests compete for user attention and computational resources; therefore, event smoothing, speculative updates, and disciplined error handling are important in the pipeline. Smoothing reduces the frequency of reactions to rapid sequences of changes, especially during text input and drag operations. Speculative updates enable the interface to display the anticipated result before the server confirms, reducing perceived latency and supporting the continuity of action. Error handling must be embedded in the interface and data structures so that every operation has a defined state and a clear recovery path. Request errors, transformation errors, and plan validation errors must differ in signals and consequences. At the same time, the generative component must not mask errors, since hidden degradation leads to a loss of trust and to the accumulation of inconsistencies between the plan and actual data.

Data–plan reconciliation is the central task, since the plan defines the structure of representation, and data defines the content and constraints. Reconciliation begins with the requirement that each visual region refer to an explicit data source and to a transformation that renders the data suitable for display. The plan must include a minimal set of dependencies so that changes to a data fragment trigger updates only for the associated nodes in the component tree. When constructing the plan, potential data incompleteness and format mismatches must be accounted for. Therefore, the specification must support empty states, latencies, and progressive loading. The generative model may propose a structure and a set of representations, but the final decision on display admissibility is determined by the component policy and the data validator. This preserves invariants when external responses change in form and timing.

Stability of user experience in a generative interface requires layout contracts and mechanisms that preserve structure under local changes. A layout contract defines a typical page schema and fixes locations for filters, main content, details, and service messages. Templates reduce the likelihood of chaotic restructurings with each query refinement and create repeatability, accelerating user learning. Stable node identifiers are required to preserve component state, since restructuring the tree without stable keys results in loss of focus, scroll resets, and the disappearance of entered values. Partial updates should be the norm, because complete regeneration makes the interface jittery and undermines the sense of control. Partial updating is achieved through differential plan comparison, immutable data models, and constrained repaint regions. Under these conditions, the generative event cycle becomes similar to a managed editor, in which the user sees the interface evolve.

Testing of a generative interface begins with fixing the form of the plan fed into the renderer. Specification snapshots are useful because they decouple checking structure from checking visual implementation and allow tracking changes in the component tree under identical inputs. Such snapshots must be constructed after normalization so that differences in field order and non-essential identifiers do not introduce noise. In addition, property-based checks are required, in which an input generator systematically enumerates combinations of data and events and verifies global invariants, including bounded tree depth, absence of forbidden properties, correct connectivity of data sources, and predictability of local state. Critical user scenarios require end-to-end tests, since it is in these scenarios that latent dependencies among state, data, and plan manifest. An end-to-end test must model the user's path through input, data loading, error handling, and return to a working state. It verifies the robustness of layout contracts and the preservation of key states under partial updates, which were discussed earlier as a condition for stable user experience.

Observability is needed to keep the generative event cycle explainable and controllable as complexity grows. Logs must record user input, the formed intention, a concise description of the plan, and the validation outcome. This binds rendering to the decision source and facilitates analysis of degradations. Metrics must reflect validation errors, the frequency of repeated renders, and generation

time, as these indicators are directly related to interaction quality and cost. Validation errors signal divergence between the model and the component policy. Frequent re-runs indicate unstable keys and violations of the partial update principles. Generation time reflects how well the system meets interactivity expectations and how justified strategic caching decisions are. For practical diagnosis, distributions of latencies across pipeline stages are also important to distinguish data retrieval latency, plan construction latency, and rendering latency.

Risks in generative interfaces arise at the boundary between natural language and executable behavior. Prompt injection exploits the fact that user text may contain instructions intended to circumvent policy or exfiltrate confidential data. Protection begins with providing the model with a strictly constrained component registry and action protocol, as discussed above. Further, input filtering and channel isolation for channels containing secrets are required to prevent these secrets from entering the generation context. Quality control relies on versioning of the plan schema and the component registry, since changes in admissible properties and actions must be reversible and auditable. Cost is determined by the frequency of model invocations and the volume of processed data. Caching at the intention–data boundary, limits on plan size and regeneration frequency, and rate limiting at the event cycle level are crucial here. These practices preserve system controllability and connect security, quality, and performance into a single policy, which remains, together with state and data, a source of truth. The Generative Interface Testing Cycle is depicted in Figure 2.

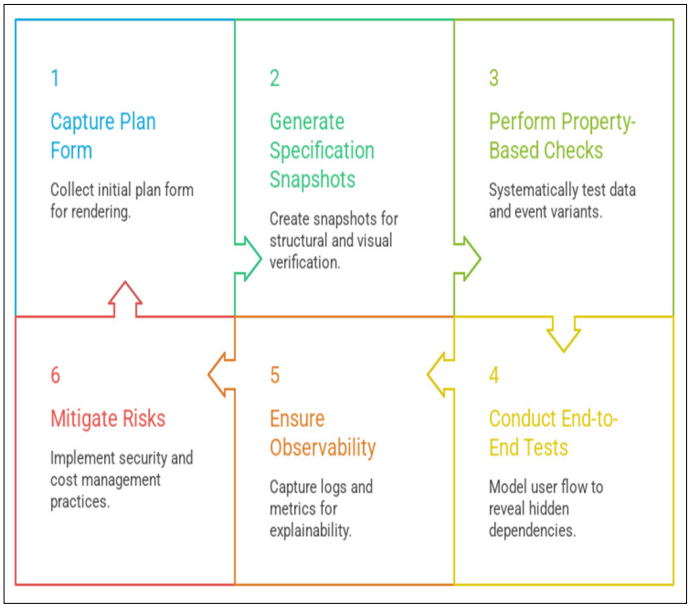

Fig. 2. Generative Interface Testing Cycle

Consider a scenario in which a user constructs a sales dashboard by formulating a task in natural language and expects to obtain a working interface for analysis. The pipeline input is a description of the goal, a list of dimensions, and constraints, such as the selection of period, region, and channel. The system extracts the intention and matches it with available data sources and admissible operations. An interface plan is then formed that includes the page frame, filter area, indicator area, tabular detail view, and graphical representation of dynamics. The plan uses only registry elements; therefore, layout components, input elements, the table, the chart, and notifications are selected from a fixed set and described via properties compatible with the design system. This means that both the empty and loading states were already designed for in our specification meaning that the interface is resilient against latency and requests not being finished.

Each node in the specification has an unchanging unique identifier, a component type, a list of properties, and references to data declarations and action declarations. Nodes are arranged into a hierarchical structure. Data sources are described as named queries with parameters dependent on the filter state. Derived fields are transformations that adapt data to table and chart formats. The component policy checks the admissibility of each node, and the specification validator confirms the consistency of types and constraints, including tree depth and collection sizes. After this, the renderer constructs the component tree and links it to the state store and data layer. Rendering is performed so that changing a filter updates only the associated regions and does not trigger a full-page reconstruction. This preserves scroll position, input focus, and observational context, which is critical for analytical behavior.

Event handling is organized around an action dispatcher, which translates interface events into declarative commands. Changing the selected period produces an action that updates the filter state. The data layer then issues a new request with updated parameters and refreshes the cache. Components that depend on the result receive new data and are partially rerendered. A request error triggers a notification and preserves the last valid state so the user can adjust parameters and retry without losing work. Upon a user request for refinement, such as segmentation, an action is initiated that passes the current state context and data metadata to the generative module. The module updates the plan by adding a new grouping or an additional visualization, while preserving identifiers of existing nodes; therefore, the interface evolves without abrupt jumps. This scenario demonstrates a closed loop in which intention governs the plan, the plan is constrained by the registry, rendering follows policy, and events reconcile data and state at an interactive pace.

CONCLUSION

The study identifies a shift from dialog assistants to interface generation as users increasingly expect systems to complete multi-step tasks with minimal mediation. In this context, linear chat becomes inefficient for navigating complex information, refining queries, and verifying results. As a

result, greater importance is assigned to interface structure, controllability, and behavioral predictability.

The analysis distinguishes two engineering trajectories. Direct generation of component source code provides flexibility, yet it raises the risk of syntactic errors, architectural inconsistency, and unintended side effects. By contrast, generation through a structured specification with schema validation improves control, restricts the space of admissible constructs, and simplifies security and design-system governance.

The concluding perspective associates robust user experience with disciplined management of state, data, and component policy. Stable interactivity depends on constrained action protocols, continuity-preserving updates, and clear differentiation of error types. Reliability is further supported by specification-based testing, observability mechanisms, and governance practices such as schema versioning, dependency control, caching, and limits on regeneration.

REFERENCES

1. Chavarriaga, E., Jurado, F., & Rodríguez, F. D. (2023). An approach to build JSON-based Domain Specific Languages solutions for web applications. *Journal of Computer Languages*, *75*, 101203. https://doi.org/10.1016/j.cola.2023.101203
2. Erazo Garzón, L., Suquisupa, S., Bermeo, A., & Cedillo, P. (2023). ModelDriven Engineering Applied to User Interfaces. A Systematic Literature Review. In M. BottoTobar, Z. Vizuete, M. L. Sergio, P. TorresCarrión, & B. Durakovic (Eds.), *Applied Technologies* (pp. 575–591). Springer Nature Switzerland.
3. Gou, Q., Dong, Y., & Ke, Q. (2024). SynthoMinds: Bridging human programming intuition with retrieval, analogy, and reasoning in program synthesis. *Journal of Systems and Software*, *216*, 112140. https://doi.org/10.1016/j.jss.2024.112140
4. Lamine, Y., & Cheng, J. (2022). Understanding and supporting the design systems practice. *Empirical Software Engineering*, *27*, 146. https://doi.org/10.1007/s10664-022-10181-y
5. Ni, A., Yin, P., Zhao, Y., Riddell, M., Feng, T., Shen, R., Yin, S., Liu, Y., Yavuz, S., Xiong, C., Joty, S., Zhou, Y., Radev, D., Cohan, A., & Cohan, A. (2024). L2CEval: Evaluating LanguagetoCode Generation Capabilities of Large Language Models. *Transactions of the Association for Computational Linguistics*, *12*, 1311–1329. https://doi.org/10.1162/tacl_a_00705
6. Ross, S. I., Martinez, F., Houde, S., Muller, M., & Weisz, J. D. (2023). The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. *Proceedings of the 28th International Conference on Intelligent User Interfaces*. https://doi.org/10.1145/3581641.3584037
7. Singh, S., & Beniwal, H. (2021). A survey on near-human conversational agents. *Journal of King Saud University - Computer and Information Sciences*, *34*(10). https://doi.org/10.1016/j.jksuci.2021.10.013
8. Tabalba, R., Kirshenbaum, N., Leigh, J., Bhatacharya, A., Johnson, A., Grosso, V., Di Eugenio, B., & Zellner, M. (2022). Articulate+: An Always-Listening Natural Language Interface for Creating Data Visualizations. *Proceedings of the 4th Conference on Conversational User Interfaces*, 1–6. https://doi.org/10.1145/3543829.3544534
9. Tan, X., Hou, B., & Zhang, L. (2026). Securing the Foundation of AI: A Deep Dive into Dependency Management and Security in Deep Learning Frameworks. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3798048
10. Wang, B., Li, G., & Li, Y. (2022). Enabling Conversational Interaction with Mobile UI using Large Language Models. *ArXiv*. https://doi.org/10.48550/arxiv.2209.08655
11. Xiao, S., Chen, Y., Song, Y., Chen, L., Sun, L., Zhen, Y., Chang, Y., & Zhou, T. (2024). UI semantic component group detection: Grouping UI elements with similar semantics in mobile graphical user interface. *Displays*, *83*, 102679. https://doi.org/10.1016/j.displa.2024.102679