EXPOZY Template JSON Schema

Abstract

This paper describes the method used to design Template Schema (JSON Schema) for an AI-powered page generator in EXPOZY, and how the existing EXPOZY front-end framework was analyzed to derive schema constraints. Because unconstrained LLM-generated HTML/JS is difficult to validate, maintain, and secure, the project adopts a structured-output approach: the AI outputs a restricted JSON template package that is validated using JSON Schema and additional semantic rules before any preview or publishing occurs. The schema design is based on reverse-engineering key patterns from the EXPOZY storefront framework, including a global reactive data object, declarative REST bindings (apiData), the universal action handler (alpineListener), SPA-like routing (href()), and TailwindCSS class usage (expozy-ui, n.d.).

---

1. Introduction

The project goal is to enable generation of EXPOZY-compatible page templates from natural language prompts, while preserving security, quality, and maintainability. The primary risk of using LLMs for direct code generation is producing output that is invalid, unsafe (e.g., XSS vectors), or inconsistent with platform conventions (OWASP Cheat Sheet Series, n.d.-a).

To address this, the project introduces Template Schema: a strict JSON Schema that defines what the AI is allowed to output. Only after schema validation and security checks does the system render a preview or allow publishing (Wright et al., 2022).

1.1 Research questions and contribution

This study investigates how an AI system can generate templates that remain compatible with EXPOZY while being validatable and safer than free-form code generation.

- **RQ1:** Which EXPOZY template behaviors and conventions must be representable in a constrained intermediate format to support real storefront pages?
- **RQ2:** How can those behaviors be encoded so that AI-generated outputs are structurally valid and safer than generating free-form HTML/JavaScript?

Contribution:

1. **Template Schema** (schema + conventions) that restricts AI output to an allowlisted template package.

2. A **reverse-engineered mapping** from EXPOZY's front-end code base.
3. A **validation approach** combining structural schema conformance with semantic validation and security boundaries before preview/publishing.

2. Research Method: How the Code Was Analyzed

This study uses a structured reverse-engineering methodology because the goal is to derive a schema from an existing implementation i.e., move from concrete code artifacts (templates, routing patterns, data-loading hooks) to an abstract representation that can be validated and regenerated. **(Chikofsky & Cross, 1990)**.

The analysis was performed as static analysis because the required evidence (file structure, template "surface area," permitted directives and endpoint patterns) can be obtained directly from source code and documentation without executing the system. This aligns with NIST's definition of a static code analyzer as a tool that analyzes source code without running it, which matches this project's repository-based inspection procedure **(National Institute of Standards and Technology, n.d.)**.

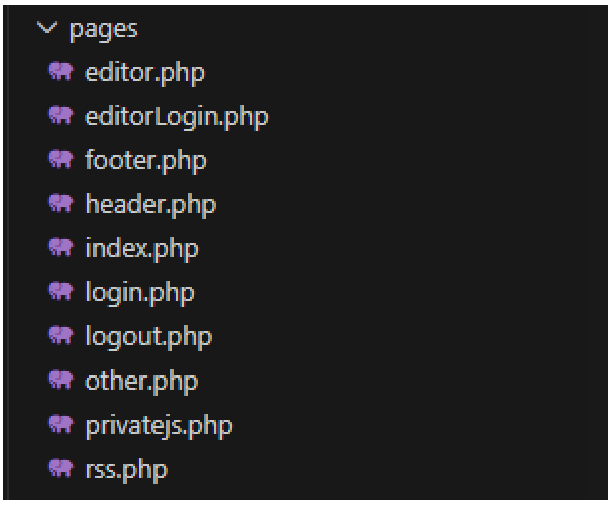
2.1 Data sources

- EXPOZY storefront framework repository used to identify conventions and constraints (expozy-ui, n.d.).
- Alpine.js directive documentation to understand and constrain reactive behavior patterns (e.g., x-data, x-text, x-for) (Alpine.js, n.d.-a; Alpine.js, n.d.-b; Alpine.js, n.d.-c).
- Tailwind CSS documentation to justify class-string constraints and the expanded output surface created by "arbitrary values" (Tailwind CSS, n.d.).
- Google Cloud Vertex AI documentation for structured output using responseMimeType and responseSchema (Google Cloud, n.d.-a; Google Cloud, n.d.-b; Google Cloud, 2025).

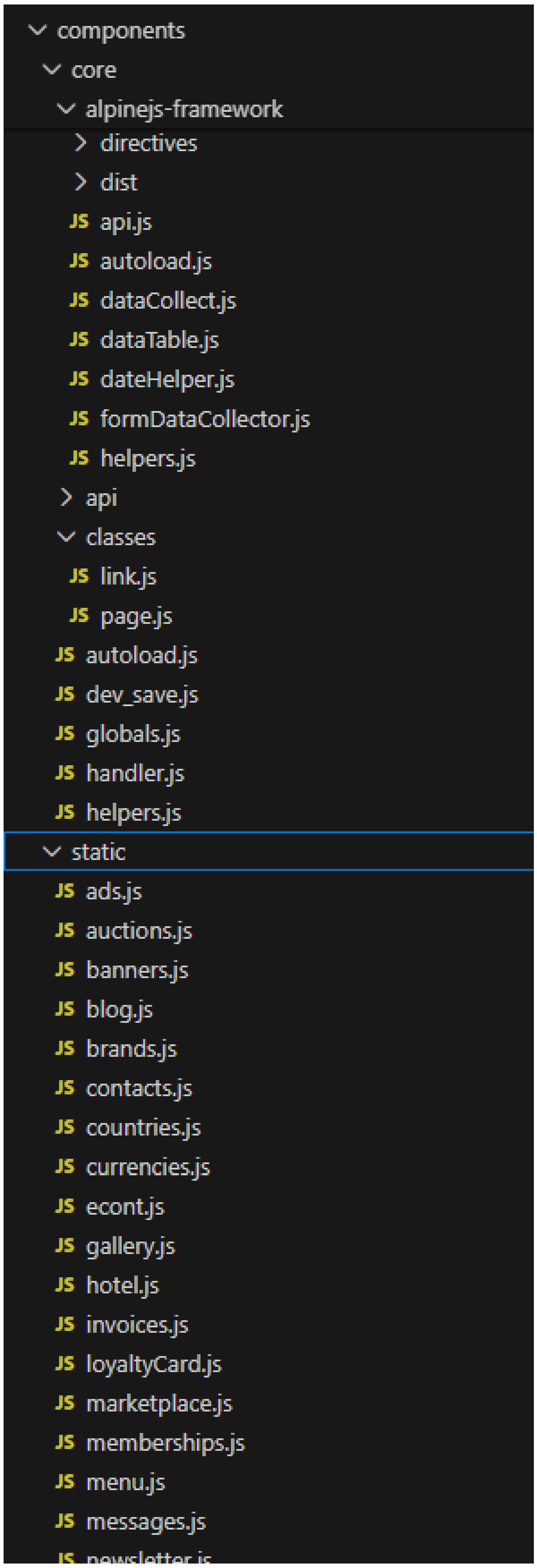
3. Static Analysis Procedure

Step A Map template

The framework was inspected to identify where templates live, how pages are composed (page shell + reusable fragments), and which page patterns recur. This establishes the minimum capability the generator must support: page metadata, layout blocks, and common section patterns used by EXPOZY templates (expozy-ui, n.d.)



```
pages
  editor.php
  editorLogin.php
  footer.php
  header.php
  index.php
  login.php
  logout.php
  other.php
  privatejs.php
  rss.php
```



```
components
  core
    alpinejs-framework
      directives
      dist
      api.js
      autoload.js
      dataCollect.js
      dataTable.js
      dateHelper.js
      formDataCollector.js
      helpers.js
    api
    classes
      link.js
      page.js
    autoload.js
    dev_save.js
    globals.js
    handler.js
    helpers.js
  static
    ads.js
    auctions.js
    banners.js
    blog.js
    brands.js
    contacts.js
    countries.js
    currencies.js
    econt.js
    gallery.js
    hotel.js
    invoices.js
    loyaltyCard.js
    marketplace.js
    memberships.js
    menu.js
    messages.js
    newsletter.js
```

Figure 1 and 2: Project structure

The figures come from the repository. They show that the main template files are in the pages/ folder. These files create the basic page “shell” and the different page endpoints. The shell is split into reusable parts (header.php and footer.php) and a middle area where the page content is loaded dynamically.

In header.php, the template sets up the <head> section (SEO tags, canonical link, favicon), defines global JavaScript variables, and connects Alpine to the page with x-data="data". It also applies dark mode by adding a class when data.darkMode is true. In footer.php, the template loads the main framework scripts (for example autoload.js and the SPA link handler link.js) and includes places where extra scripts/styles can be added.

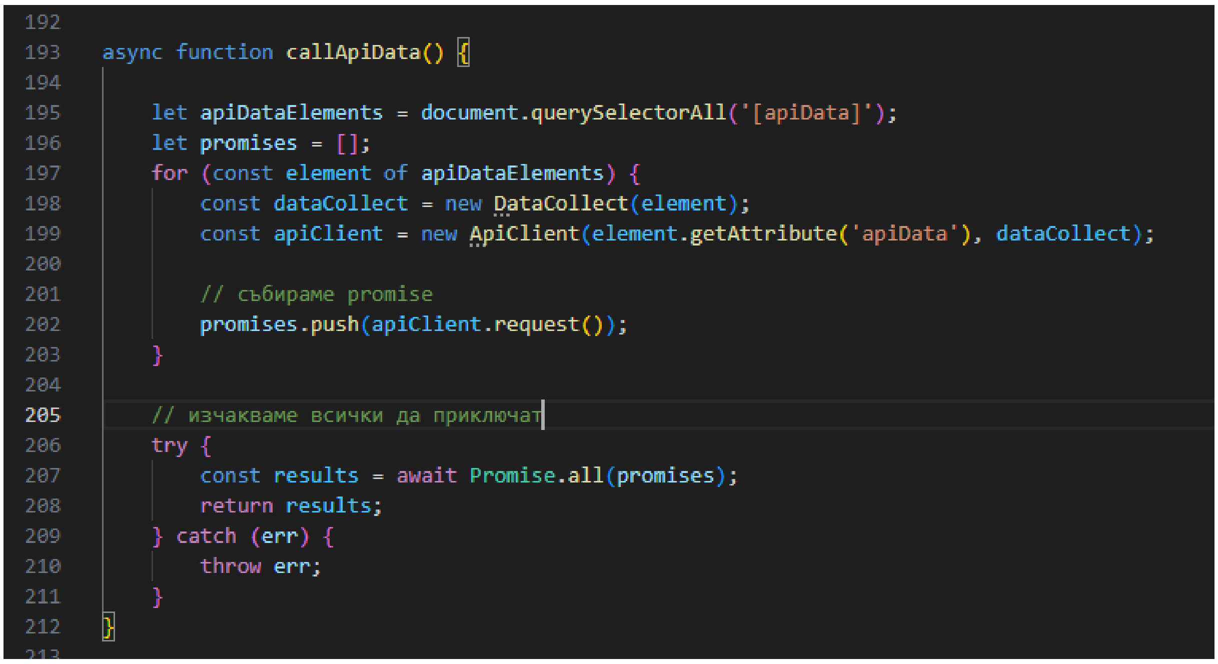
The page is built using a slot system: the header, main content, and footer are inserted from $page->header, $page->html, and $page->footer using Page::html_res_change(...). This means the PHP files mostly provide a stable layout wrapper, while the actual page content is stored separately as editable HTML fragments. So, the template “surface area” includes: (1) fixed layout slots (header/main/footer), (2) shared UI features like notifications, theme switching, and script injection, and (3) specific endpoints such as login/logout pages, editor/admin pages, and non-visual endpoints like RSS or private content loading.

Step B Inventory allowed behaviors

List what EXPOZY templates are allowed to do in markup, so the schema can represent the same behaviors in a controlled way. This step focuses on: how templates load data, how templates run actions when users click buttons, and what endpoint formats are allowed (expozy-ui, n.d.).

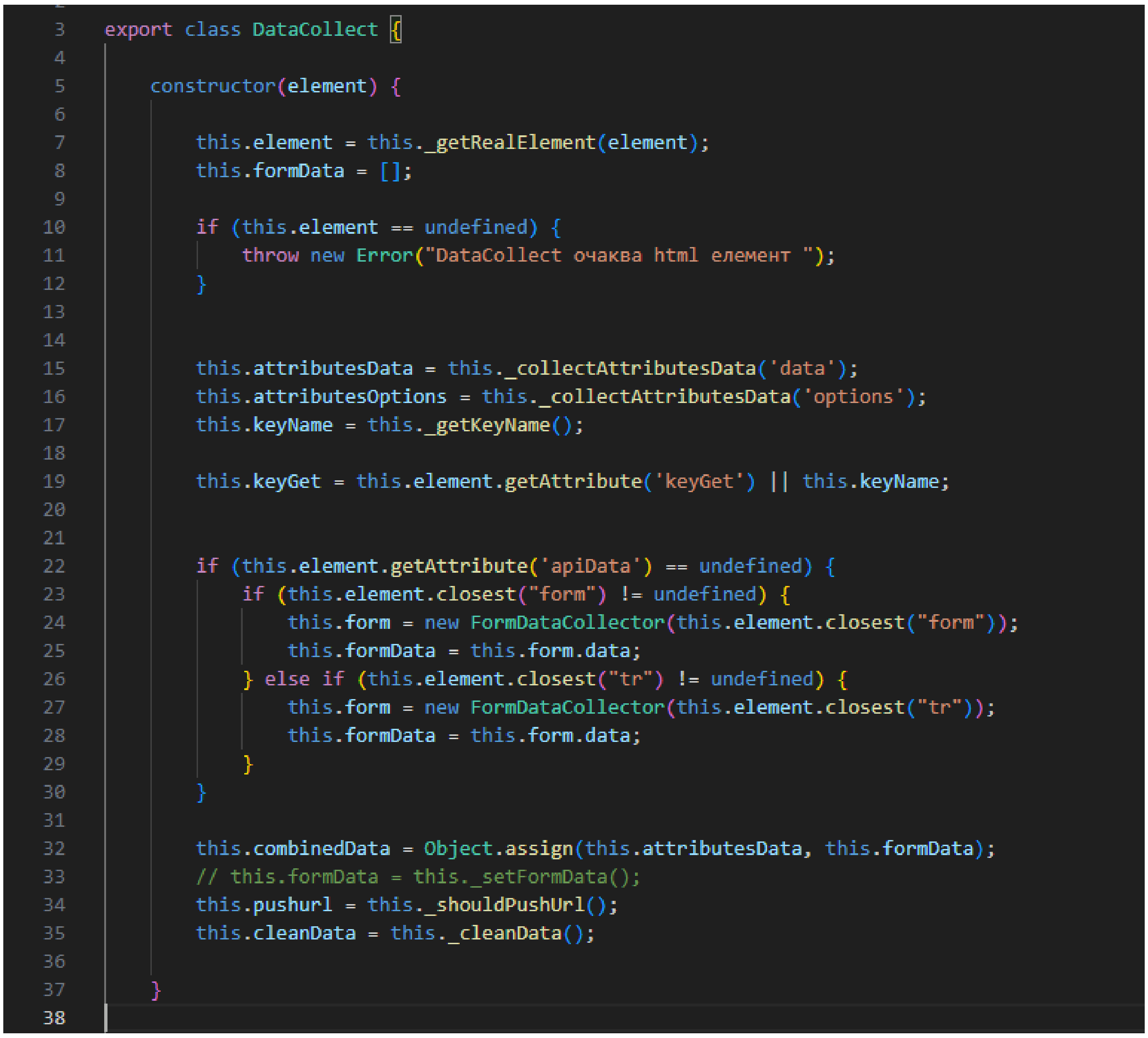
B1) Allowed data loading

EXPOZY supports data loading through HTML attributes. Elements with apiData="..." are discovered by callApiData() (which scans the DOM for [apiData]) and can be executed automatically during page initialization. Request parameters are passed using data-* attributes (e.g., data-limit="12") and are collected into a request object. Templates can also provide keyName="...", which is forwarded as an option (options.keyName) so the called module can store the response under the matching key in the global data store (e.g., dataProxy[keyName]). The framework also supports options-* flags (confirmed in this code for options-pushurl), which modify runtime behavior after the request (expozy-ui, n.d.).
(Examples from the code are shown below for callApiData() and DataCollect().)



```
async function callApiData() {

    let apiDataElements = document.querySelectorAll('[apiData]');
    let promises = [];
    for (const element of apiDataElements) {
        const dataCollect = new DataCollect(element);
        const apiClient = new ApiClient(element.getAttribute('apiData'), dataCollect);

        // събираме promise
        promises.push(apiClient.request());
    }

    // изчакваме всички да приключат
    try {
        const results = await Promise.all(promises);
        return results;
    } catch (err) {
        throw err;
    }
}
```

Figure 3 callApiData()



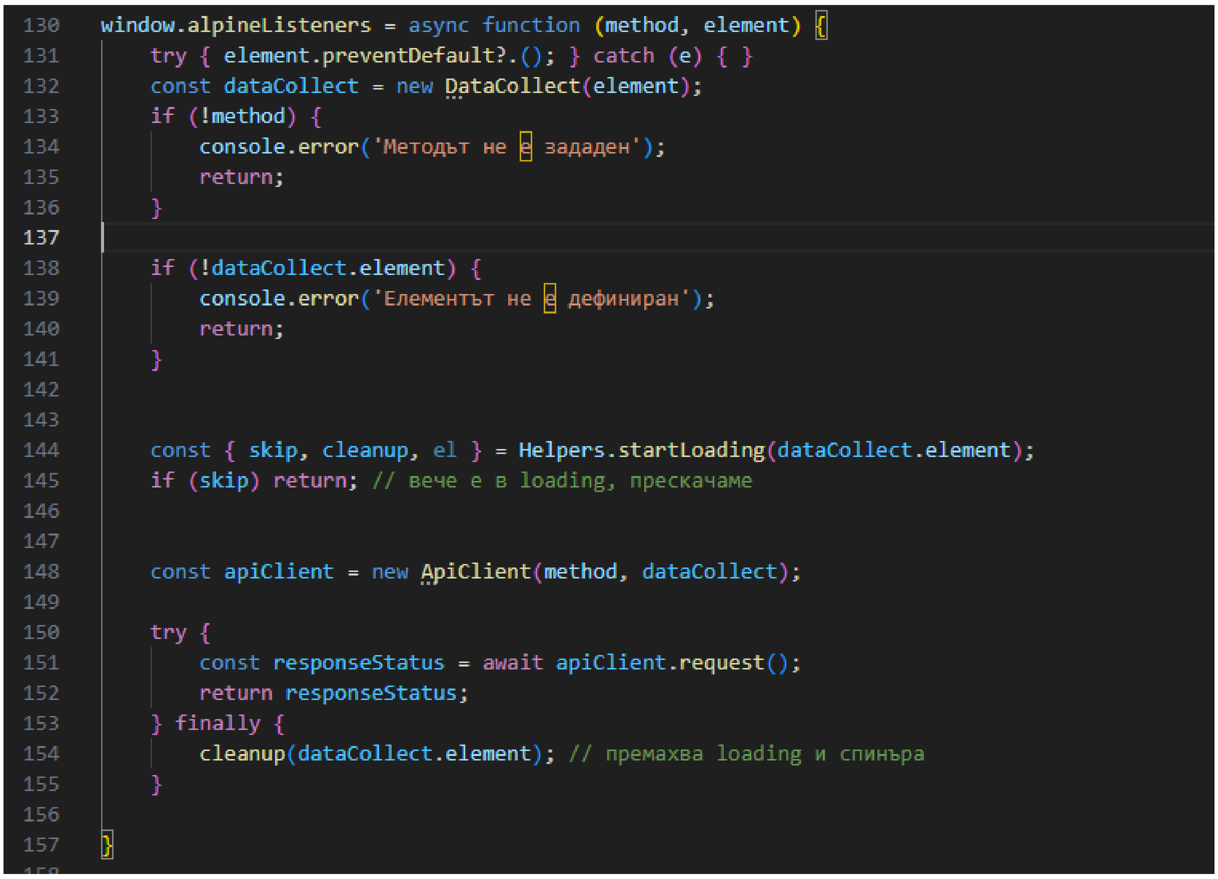
```
export class DataCollect {

    constructor(element) {

        this.element = this._getRealElement(element);
        this.formData = [];

        if (this.element == undefined) {
            throw new Error("DataCollect очаква html елемент ");
        }


        this.attributesData = this._collectAttributesData('data');
        this.attributesOptions = this._collectAttributesData('options');
        this.keyName = this._getKeyName();

        this.keyGet = this.element.getAttribute('keyGet') || this.keyName;


        if (this.element.getAttribute('apiData') == undefined) {
            if (this.element.closest("form") != undefined) {
                this.form = new FormDataCollector(this.element.closest("form"));
                this.formData = this.form.data;
            } else if (this.element.closest("tr") != undefined) {
                this.form = new FormDataCollector(this.element.closest("tr"));
                this.formData = this.form.data;
            }
        }

        this.combinedData = Object.assign(this.attributesData, this.formData);
        // this.formData = this._setFormData();
        this.pushurl = this._shouldPushUrl();
        this.cleanData = this._cleanData();

    }

```

Figure 4 DataCollect()

B2) Allowed actions (alpineListeners, endpoint strings, success/error)

User interactions are handled through one standard function: alpineListeners(method, element). Templates do not write custom request code. Instead, they call alpineListeners() with an endpoint string (example: Shop.post_carts) and the clicked element. The framework collects inputs from the element using data-* attributes (and form inputs when needed), runs the request, and can store the result into data[keyName]. The framework also reports results using success/error handling (expozy-ui, n.d.).

(Examples from the code are shown in Section 3.3: alpineListeners())



```
window.alpineListeners = async function (method, element) {
    try { element.preventDefault?.(); } catch (e) { }
    const dataCollect = new DataCollect(element);
    if (!method) {
        console.error('Методът не е зададен');
        return;
    }

    if (!dataCollect.element) {
        console.error('Елементът не е дефиниран');
        return;
    }


    const { skip, cleanup, el } = Helpers.startLoading(dataCollect.element);
    if (skip) return; // вече е в loading, прескачаме


    const apiClient = new ApiClient(method, dataCollect);

    try {
        const responseStatus = await apiClient.request();
        return responseStatus;
    } finally {
        cleanup(dataCollect.element); // премахва loading и спинъра
    }

}
```

Figure 5: alpineListeners

B3) Endpoint formats

EXPOZY uses two endpoint formats:

- **API-style:** {verb}.{resource} = get.products
- **Module-style:** {Module}.{method} = Shop.post_carts

This makes it possible to validate endpoints in the schema and block unknown or unsafe calls (expozy-ui, n.d.).

Step C Identify security-relevant boundaries
Determine where HTML injection could occur and enforce sanitization/allowlisting. OWASP recommends applying appropriate defenses against XSS and emphasizes using multiple defensive techniques where needed (OWASP Cheat Sheet Series, n.d.-a). OWASP also recommends input validation approaches such as allowlisting and validating early in the data flow (OWASP Cheat Sheet Series, n.d.-b). This is part of the validation layer in the back-end of expozy because it cannot be implemented through vertex AI schema .

Step D Convert findings into a constrained intermediate representation
Replace "free-form HTML + free-form Alpine expressions" with:

- Component types (enum)
- Typed props
- Declared dataSources
- Declared actions
- Restricted directives
- Controlled Tailwind class strings
  This conversion is the core step that turns a codebase into a schema-driven generator.

4. Schema design (Template Schema v1)

4.1 Design goals

The Template Schema was designed to ensure structural validity, meaning the model output is always well-formed JSON that conforms to a predefined shape enforced through schema-constrained generation (Google Cloud, n.d.-a). It also targets platform compatibility by representing the same core behaviors used in EXPOZY templates, such as declarative data loading, standardized actions, and consistent page structure conventions (expozy-ui, n.d.). In addition, the schema supports security goals by limiting unsafe output patterns and requiring sanitization for any rich text or HTML-like content before rendering, following established guidance for reducing XSS risk (OWASP Cheat Sheet Series, n.d.). Finally, the approach improves maintainability by favoring stable, reusable building blocks over unconstrained, arbitrary code generation.

4.2 Top-level structure

A template package is organized into:

- metadata (id, name, pageType, route, SEO fields)
- dataSources[] (declared data requirements)
- actions[] (declared interactions)
- layout or sections[] (allowlisted UI building blocks)

4.3 Alpine directive policy (behavior constraints)

Alpine enables reactive UI directly in markup (e.g., component state via x-data, rendering via x-text, loops via x-for). Template Schema therefore applies an allowlist approach: only a limited set of directive patterns are permitted, while risky patterns (such as raw HTML insertion without sanitization controls) are restricted and handled through the validation layer (Alpine.js, n.d.-a; Alpine.js, n.d.-b; Alpine.js, n.d.-c; OWASP Cheat Sheet Series, n.d.).

4.4 Tailwind policy (style constraints)

Tailwind supports “arbitrary values” and “arbitrary variants,” which increases expressiveness and expands the space of possible generated outputs. The schema therefore treats class strings as data with constraints (e.g., length/character bounds and allowlisting rules enforced during validation) (Tailwind CSS, n.d.).

5. Validation and controls

5.1 Structural conformance (model-level)

In implementation, structural validity is enforced using Vertex AI structured output by setting responseMimeType to application/json and providing a responseSchema. Vertex AI documents that responses can be constrained to always follow the provided schema, enabling reliably structured JSON output (Google Cloud, n.d.-a). Vertex AI further specifies that responseSchema is based on a subset of the OpenAPI 3.0 schema object, and requires responseMimeType to be set to application/json when responseSchema is used (Google Cloud, n.d.-b; Google Cloud, 2025).

5.2 Semantic validation (platform-level)

Some EXPOZY requirements cannot be expressed purely as a response schema (for example: verifying that references point to existing actions/data sources, enforcing endpoint allowlists per tenant, and preventing key collisions). These checks are handled as procedural validation in the backend (expozy-ui, n.d.).

5.3 Sanitization boundaries (security-level)

If rich text or HTML-like content is allowed anywhere in the template package, it must be sanitized before rendering. OWASP emphasizes layered controls and careful handling of untrusted content to reduce XSS risk (OWASP Cheat Sheet Series, n.d.). This will be done in the code of the system itself.

6. Template Schema

This is the result of the research

{ "type": "OBJECT", "properties": { "metadata": { "type": "OBJECT", "properties": { "id": { "type": "STRING" }, "name": { "type": "STRING" }, "pageType": { "type": "STRING", "enum": ["landing", "product", "category", "blog", "cart", "account", "contact", "custom"] }, "route": { "type": "STRING", "nullable": true }, "title": { "type": "STRING", "nullable": true }, "description": { "type": "STRING", "nullable": true } }, "required": ["id", "name", "pageType"], "propertyOrdering": ["id", "name", "pageType", "route", "title", "description"] }, "theme": { "type": "OBJECT", "nullable": true, "properties": { "primaryColor": { "type": "STRING",

"nullable": true }, "darkMode": { "type": "BOOLEAN", "nullable": true } },
"propertyOrdering": ["primaryColor", "darkMode"] }, "dataSources": { "type": "ARRAY",
"nullable": true, "items": { "type": "OBJECT", "properties": { "id": { "type": "STRING" },
"endpoint": { "type": "STRING" }, "keyName": { "type": "STRING" }, "params": { "type":
"OBJECT", "nullable": true, "properties": { "limit": { "type": "INTEGER", "nullable": true },
"page": { "type": "INTEGER", "nullable": true }, "filter": { "type": "STRING", "nullable": true },
"category_id": { "type": "STRING", "nullable": true }, "product_id": { "type": "STRING",
"nullable": true }, "slug": { "type": "STRING", "nullable": true }, "sort": { "type": "STRING",
"nullable": true }, "order": { "type": "STRING", "nullable": true }, "search": { "type":
"STRING", "nullable": true }, "status": { "type": "STRING", "nullable": true }, "tag": { "type":
"STRING", "nullable": true } }, "propertyOrdering": ["limit", "page", "filter", "category_id",
"product_id", "slug", "sort", "order", "search", "status", "tag"] }, "options": { "type":
"OBJECT", "nullable": true, "properties": { "pushurl": { "type": "BOOLEAN", "nullable": true
}, "scroll": { "type": "BOOLEAN", "nullable": true }, "clear": { "type": "BOOLEAN",
"nullable": true } }, "propertyOrdering": ["pushurl", "scroll", "clear"] }, "autoLoad": {
"type": "BOOLEAN", "nullable": true } }, "required": ["id", "endpoint", "keyName"],
"propertyOrdering": ["id", "endpoint", "keyName", "params", "options", "autoLoad"] } },
"actions": { "type": "ARRAY", "nullable": true, "items": { "type": "OBJECT", "properties": {
"id": { "type": "STRING" }, "endpoint": { "type": "STRING" }, "keyName": { "type": "STRING",
"nullable": true }, "options": { "type": "OBJECT", "nullable": true, "properties": { "pushurl":
{ "type": "BOOLEAN", "nullable": true }, "scroll": { "type": "BOOLEAN", "nullable": true },
"clear": { "type": "BOOLEAN", "nullable": true } }, "propertyOrdering": ["pushurl", "scroll",
"clear"] } }, "required": ["id", "endpoint"], "propertyOrdering": ["id", "endpoint",
"keyName", "options"] } }, "sections": { "type": "ARRAY", "items": { "type": "OBJECT",
"properties": { "id": { "type": "STRING", "nullable": true }, "type": { "type": "STRING",
"enum": ["hero", "content", "products", "posts", "form", "cta", "features", "testimonials",
"faq", "footer"] }, "className": { "type": "STRING", "nullable": true }, "title": { "type":
"STRING", "nullable": true }, "subtitle": { "type": "STRING", "nullable": true }, "content": {
"type": "STRING", "nullable": true }, "backgroundImage": { "type": "STRING", "nullable":
true }, "dataSource": { "type": "STRING", "nullable": true }, "actionRef": { "type": "STRING",
"nullable": true }, "columns": { "type": "INTEGER", "nullable": true }, "items": { "type":
"ARRAY", "nullable": true, "items": { "type": "OBJECT", "properties": { "title": { "type":
"STRING", "nullable": true }, "content": { "type": "STRING", "nullable": true }, "icon": {
"type": "STRING", "nullable": true }, "href": { "type": "STRING", "nullable": true }, "image": {
"type": "STRING", "nullable": true } }, "propertyOrdering": ["title", "content", "icon", "href",
"image"] } }, "buttons": { "type": "ARRAY", "nullable": true, "items": { "type": "OBJECT",
"properties": { "text": { "type": "STRING" }, "href": { "type": "STRING", "nullable": true },
"actionRef": { "type": "STRING", "nullable": true }, "variant": { "type": "STRING", "enum": [
"primary", "secondary", "outline"], "nullable": true } }, "required": ["text"],
"propertyOrdering": ["text", "href", "actionRef", "variant"] } }, "fields": { "type": "ARRAY",
"nullable": true, "items": { "type": "OBJECT", "properties": { "name": { "type": "STRING" },

"label": { "type": "STRING", "nullable": true }, "type": { "type": "STRING", "enum": ["text", "email", "password", "textarea", "select", "checkbox"], "nullable": true }, "placeholder": { "type": "STRING", "nullable": true }, "required": { "type": "BOOLEAN", "nullable": true } }, "required": ["name"], "propertyOrdering": ["name", "label", "type", "placeholder", "required"] } } }, "required": ["type"], "propertyOrdering": ["id", "type", "className", "title", "subtitle", "content", "backgroundImage", "dataSource", "actionRef", "columns", "items", "buttons", "fields"] } } }, "required": ["metadata", "sections"], "propertyOrdering": ["metadata", "theme", "dataSources", "actions", "sections"] }

To be able to view it properly use JSON Beautify - JSON Formatter and JSON Validator Online

To validate whether the schema works, tests are used that evaluate AI-generated template packages against structural, semantic, and security constraints derived from the EXPOZY framework.

7. Evaluation

To evaluate whether Template Schema achieves its goals (validity, EXPOZY compatibility, safety, and policy compliance), a prompt-based test suite was executed using the same structured-output configuration used in development (responseMimeType = application/json with a responseSchema). The suite intentionally mixes (1) representative EXPOZY page types, (2) edge cases that stress schema limits, and (3) adversarial probes designed to trigger the most common failure modes identified in the research (unsafe markup, invalid endpoints, broken references, and policy violations).

Test design. The prompts cover the core page categories defined in the schema (landing, product, category, blog, cart, account, contact, and custom). They also include targeted prompts for schema features such as dataSources options (pushurl/scroll/clear, autoLoad), multiple actions in a single page, and coverage of all section types. Edge-case prompts test minimal content (e.g., a hero with only one button), long content blocks (3–5 paragraphs), full form field coverage (text/email/password/textarea/select/checkbox), and button variants (primary/secondary/outline). Finally, adversarial probes attempt to introduce XSS-like payloads (script tags, javascript: URLs, inline event handlers, iframes), unsafe Alpine patterns (x-html, x-init/fetch/eval, function-call handlers), unsafe Tailwind arbitrary-content patterns (e.g., content-[...] with HTML), suspicious theme values, and risky routes (query strings or script-like characters).

Each generated output was scored using the following criteria derived directly from the project requirements and validation layers:

Parse success rate: percentage of cases where the model output is valid JSON that can be parsed successfully. This is a prerequisite for any further validation or rendering and supports the structured-output goal in Section 5.1.

Schema adherence rate: percentage of cases where the parsed output conforms to the Template Schema structure (required fields, types, and enums) using a Vertex-style schema validator. This corresponds to structural conformance (Section 5.1).

Endpoint validity rate: percentage of cases where all declared endpoints in dataSources[] and actions[] match EXPOZY's allowed endpoint formats (API-style {verb}.{resource} or module-style {Module}.{method}) and do not match configured "dangerous" endpoint heuristics. This supports the endpoint rules derived from reverse-engineering (Step B3) and enforced as semantic validation (Section 5.2).

Cross-reference validity rate: percentage of cases where references used in sections/buttons (e.g., dataSource, actionRef) resolve to existing declared IDs in dataSources[] and actions[]. This measures semantic correctness beyond structure and corresponds to Section 5.2.

Security clean rate: percentage of cases without matches to known unsafe patterns such as <script>, javascript:, inline event handlers (on*=), and embedded objects/iframes. This is a heuristic indicator that complements server-side sanitization described in Section 5.3.

Alpine compliance rate: percentage of cases without restricted Alpine patterns (e.g., x-html, x-init with fetch/eval, or handler strings containing function calls). This measures adherence to the Alpine directive policy and behavioral constraints (Section 4.3).

Tailwind compliance rate: percentage of cases where section className strings meet policy constraints (length limits and no dangerous arbitrary-content patterns). Arbitrary Tailwind values are allowed but flagged for review, while patterns that embed unsafe content are treated as failures. This corresponds to the Tailwind policy described in Section 4.4.

Theme validity rate: percentage of cases where theme fields are well-typed and primaryColor uses an approved color format (hex), with non-hex formats treated as warnings or errors depending on severity. This supports consistency and reduces invalid configuration at render-time.

Route validity rate: percentage of cases where metadata.route follows safe route conventions (starts with / and contains no injection-like patterns such as <script>, javascript:, or traversal sequences). This supports platform routing safety and prevents malformed navigation entries.

In addition to hard-failure criteria above, the evaluation also records **quality warnings** (not counted as failures) for page-type recommendations (e.g., product pages typically include product identifiers and a products section) and completeness issues (e.g., empty sections array, products/posts section missing a dataSource, or form sections lacking fields/actionRef). These warnings help diagnose “valid-but-unusable” outputs without conflating them with strict conformance and security failures.

8. Evaluation results and improvement plan (endpoint failures and schema guidance)

Current results (what they mean)

The evaluation executed **40 prompt-based test cases** covering page types, edge cases, and adversarial probes. All outputs were successfully parsed as JSON (**parse_success_rate = 1.0**) and conformed structurally to the response schema (**schema_adherence = 1.0**). This indicates that schema-constrained generation is effective at ensuring valid template packages.

However, the **overall error-free rate was 0.225 (9/40)**. The main contributor to failures was endpoint formatting: **endpoint_validity = 0.475 (19/40)**. In failing cases, the model produced REST-style URLs (e.g., /api/products, /api/newsletter/subscribe) rather than EXPOZY’s expected dot-notation endpoint formats (e.g., get.products or Shop.post_carts). Since endpoint checks are treated as a hard semantic constraint, these outputs were marked as errors even when they were structurally valid.

Other validation categories performed strongly:

- **crossref_validity = 0.925** shows most dataSource and actionRef references resolved correctly.
- **security_clean = 0.85** reflects that most outputs avoided unsafe strings; failures primarily occurred in intentionally adversarial security-probe tests.
- **alpine_compliance = 0.975**, **tailwind_compliance = 0.95**, **theme_validity = 0.95**, and **route_validity = 0.95** show that most outputs remained within the intended policy constraints, with only a small number of cases producing disallowed patterns.

Root cause analysis (why endpoint validity is low)

The endpoint failures are explained by a mismatch between (1) what the schema *structurally* allows and (2) what the platform *semantically* requires.

The current response schema defines endpoint fields only as a generic string ("type": "STRING"), which provides **no model guidance** about required formats. As a result, the model defaults to a common web development convention REST paths such as /api/...even though EXPOZY uses a different convention (verb.resource and Module.method). This is consistent with the failure logs, where most errors are of the form:

- $.dataSources[i].endpoint '/api/...' is not an allowed pattern
- $.actions[i].endpoint '/api/...' is not an allowed pattern

In other words: **the structured output is valid JSON, but it is not aligned with EXPOZY’s endpoint conventions**, because the schema did not communicate those conventions strongly enough.

To address the observed endpoint mismatch, the next iteration of Template Schema adds **descriptive constraints** to the response schema using description fields. Vertex AI uses these descriptions as guidance during generation, so this change improves compliance without changing the overall schema shape.

The planned changes are:

1. **Endpoint format guidance**
   Add explicit instructions to the endpoint fields in both dataSources[] and actions[] stating that endpoints must be in EXPOZY dot-notation formats:
   - API-style: verb.resource (e.g., get.products, get.posts)
   - Module-style: Module.method (e.g., Shop.post_carts)
     and explicitly disallow /api/... URL paths.
     Expected impact: increase endpoint_validity substantially, since most failures are caused by REST-style URL generation.
2. **Cross-reference guidance (reduce remaining crossref errors)**
   Add descriptions clarifying that dataSource and actionRef must reference IDs declared in dataSources[].id and actions[].id.
   Expected impact: eliminate rare "unknown id" issues caused by the model inventing references.
3. **Security guidance in text fields (improve robustness under adversarial prompts)**
   Add descriptions in title/subtitle/content/href/className fields instructing the model to avoid scripts, javascript: URLs, and inline event handlers.
   Expected impact: reduce accidental security flags in non-security prompts, while security-probe prompts remain useful for confirming detection.
4. **Route + theme guidance (reduce formatting drift)**
   Add descriptions clarifying:
   - routes must be clean paths starting with / and contain no query strings,
   - theme colors should be provided in hex format.
     Expected impact: reduce the few remaining theme/route warnings.

Refrences:

Ajv. (n.d.-a). JSON Schema. Retrieved January 10, 2026, from https://ajv.js.org/json-schema.html

Alpine.js. (n.d.-a). x-data. Retrieved January 10, 2026, from https://alpinejs.dev/directives/data

Alpine.js. (n.d.-b). x-for. Retrieved January 10, 2026, from https://alpinejs.dev/directives/for

Alpine.js. (n.d.-c). x-text. Retrieved January 10, 2026, from https://alpinejs.dev/directives/text

expozy-ui. (n.d.). Alpine-Expozy-StoreFront [Source code]. GitHub. Retrieved January 10, 2026, from https://github.com/expozy-ui/Alpine-Expozy-StoreFront

OWASP Cheat Sheet Series. (n.d.-a). Cross Site Scripting Prevention Cheat Sheet. Retrieved January 10, 2026, from https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP Cheat Sheet Series. (n.d.-b). Input Validation Cheat Sheet. Retrieved January 10, 2026, from https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

Tailwind CSS. (n.d.). Styling with utility classes. Retrieved January 10, 2026, from https://tailwindcss.com/docs/styling-with-utility-classes

Wright, A., Andrews, H., Hutton, B., & Dennis, G. (2022, June 16). Draft 2020-12. JSON Schema. Retrieved Janua

National Institute of Standards and Technology. (n.d.). *Static code analyzer*. **NIST Computer Security Resource Center (CSRC) Glossary**. Retrieved January 12, 2026. https://csrc.nist.gov/glossary/term/static_code_analyzer

Chikofsky, E. J., & Cross, J. H., II. (1990). *Reverse engineering and design recovery: A taxonomy*. **IEEE Software, 7**(1), 13–17. doi:10.1109/52.43044 https://dl.acm.org/doi/10.1109/52.43044

Alpine.js. (n.d.-a). x-data. Alpine.js documentation. https://alpinejs.dev/directives/data

Alpine.js. (n.d.-b). x-text. Alpine.js documentation. https://alpinejs.dev/directives/text

Alpine.js. (n.d.-c). x-for. Alpine.js documentation. https://alpinejs.dev/directives/for

Chikofsky, E. J., & Cross, J. H., II. (1990). Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1), 13–17. https://doi.org/10.1109/52.43044

expozy-ui. (n.d.). Alpine-Expozy-StoreFront [Source code]. GitHub. https://github.com/expozy-ui/Alpine-Expozy-StoreFront

Google Cloud. (n.d.-a). Structured output. Generative AI on Vertex AI documentation. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/multimodal/control-generated-output

Google Cloud. (n.d.-b). GenerationConfig (REST reference, v1beta1). Vertex AI documentation. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/reference/rest/v1beta1/GenerationConfig

Google Cloud. (2025, September 25). Schema (REST reference, v1beta1). Vertex AI documentation. https://docs.cloud.google.com/vertex-ai/docs/reference/rest/v1beta1/Schema

National Institute of Standards and Technology. (n.d.). Static code analyzer. NIST Computer Security Resource Center Glossary. https://csrc.nist.gov/glossary/term/static_code_analyzer

OWASP Cheat Sheet Series. (n.d.). Cross Site Scripting Prevention Cheat Sheet. OWASP. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP Cheat Sheet Series. (n.d.-b). Input Validation Cheat Sheet. OWASP. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

Tailwind CSS. (n.d.). Adding custom styles. Tailwind CSS documentation. https://tailwindcss.com/docs/adding-custom-styles