ENGINEERING REPORT · SILVERBREEZE-LABS WORKSPACE

One Chassis, Five Brands: What Code Reuse Actually Looks Like in Symbols

An honest, numbers-first review of reusability, design-system leverage, and serializable context across seven production projects built on the Symbols / DOMQL framework — including the parts that aren't perfect yet.

Workspace: `~/www/templates/silverbreeze-labs` · Framework: Symbols (DOMQL v3.14)
Generated 11 June 2026 · All metrics measured directly from the repository

1 · Executive Summary

The silverbreeze-labs workspace runs **six full applications and a widget builder** on a shared core: a 137-component default library, a 20-component domain library (`chatbot`), and a token-based design system. The leverage is measurable and unusual:

7–8
local components per consumer app — everything else is inherited

~300
lines of design-system overrides to fully re-brand an app

2 days
between the first app trio (Apr 21) and the next two full apps (Apr 23)

1 JSON
an entire app compiles to a single ~620 KB serializable document

The verdict up front: the reusability and design-system mechanics in Symbols deliver real, measured leverage that a conventional React/npm setup gets only with significant discipline and infrastructure. But the workspace also shows honest friction — duplicated navigation components that have drifted apart, design-token rules violated in places, and shared page templates that exist but aren't consumed yet. Both halves of that story are documented below.

2 · The Workspace at a Glance

Project	Role	Local components	Pages	App LOC	Inherits
libraries/default	Shared base library (`system/default`)	137	—	3,315	—
chatbot	App *and* shared domain library	20	8	2,910	default
benefits	Venue-benefits assistant	7	10	8,651	default + chatbot
greenig	Retail assistant	8	33	15,961	default + chatbot
hotcard	Customer-support assistant	7	19	14,041	default + chatbot
tetri-kvadrati	Retail assistant	8	33	16,141	default + chatbot
uhy	Support assistant (hotcard derivative)	7	19	14,661	default + chatbot
dashboard	Standalone admin product	57	39	25,508	default

LOC excludes `node_modules`, audit snapshots, and local lib caches. Component counts exclude `index.js`.

Read the component column again: **five customer-facing applications each carry only 7–8 local components**. Their entire remaining UI surface — buttons, fields, avatars, dialogs, cards, chat bubbles, admin layouts — resolves from the two shared tiers. The one app that doesn't lean on the chatbot library (dashboard) carries 57 local components, which is roughly what every app would look like without the sharing mechanism. That delta — 57 versus 7 — is the reuse dividend, visible in a single `ls`.

3 · The Story: How Five Apps Grew From One

The git history tells the story plainly. On **April 21, 2026**, three projects land together: `chatbot` (the extracted domain library), `benefits`, and `hotcard`. Two days later, on **April 23**, two more complete applications appear — `greenig` and `tetri-kvadrati`, each with 33 pages. `uhy` followed as a derivative of hotcard.

Standing up a 19–33 page branded application in days is not a normal cadence. It was possible because a new app in this workspace is, concretely:

- a `symbols.json` declaring two shared libraries (12 lines),
- a `sharedLibraries.js` wiring them at boot (4 lines),
- a ~300-line design-system layer (brand colors, semantic themes, typography),
- pages that compose inherited components by bare key — `TopBar: {}`, `BottomBar: {}`, `Chatbot: {}` — plus whatever is genuinely unique to that product.

> **The load-bearing idea:** components are referenced by *name*, not by *import path*. DOMQL resolves every bare key through a three-tier merged context — app-local first, then shared libraries in declaration order, then framework built-ins. The app never writes an import statement to use a shared component, and the local tier always wins on collision, so customization never requires touching shared source.

```
// The entire "dependency management" surface of an app — benefits/symbols.json
"sharedLibraries": {
  "system/default":            { "link": "../../libraries/default" },
  "silverbreeze-labs/chatbot": { "link": "../chatbot" }
}

// Consuming a shared component: no import, no package, no version bump
TopBar: {},          // resolves from shared tier
Chatbot: {},         // resolves from chatbot library

// Customizing it for one app: define locally, the merger picks yours
export const Card = {
  extends: 'Card',          // shared Card as base
  borderRadius: 'A2'        // override only what differs
}
```

4 · Design System: Rebranding for the Cost of a Config File

Every app ships the same 17 design-system facets (color, theme, typography, spacing, shape, animation, media queries…), but each app's local layer is thin — measured: **benefits 272 lines, greenig 280, tetri-kvadrati 286, dashboard 333, hotcard 367, uhy 403**. That thin layer is the complete visual identity of the app.

Two design decisions make this work:

Semantic themes, not colors in components

Components say `theme: 'panel'`, `theme: 'nav'`, `theme: 'document'` — never a color. Each app's `theme.js` defines what those words mean, including `@dark` / `@light` variants in one place. Dark mode is not a feature anyone implements per component; it is a property of the theme definition.

Tokens as the framework's native vocabulary

Spacing is `padding: 'Z B'`, sizing is `maxWidth: 'I1'`, colors are `'brand.08'` or `'neutral700'` with shade modifiers computed by the framework. There is no translation layer between "design system" and "code" — the tokens *are* the styling API. Compare this to a typical React stack, where the design system is a separate package, a ThemeProvider, a Tailwind config, and a lint rule, all maintained in parallel with the components.

> **Measured consequence:** the difference between hotcard (red support brand) and greenig (green retail brand) — different products, different look — is dominated by a few hundred lines of token overrides plus product-specific pages. The 137-component base library and the 20-component chatbot library are byte-identical across all consumers.

5 · Context: The Whole App Is a Document

The third pillar is less visible than reuse or tokens but arguably the most strategic. Because Symbols components are **plain serializable objects** — no classes, no JSX, no closures (the `frank` compiler enforces this) — an entire application compiles to a single JSON document. Measured on benefits: **~620 KB for the whole app** — every component, page, function, and design-system token in one tree.

What that buys, concretely, in this workspace:

- **Platform sync.** `smbls publish` pushes the document to the cloud; the visual editor and the runtime consume the same artifact. There is no "design tool version" vs "code version" drift, because there is one representation.
- **AI-native context.** An AI agent can hold an *entire application* — not a file at a time — in a single structured payload, reason about it, and emit valid patches. Each app additionally ships an `AI.md` documenting its AI architecture. This workspace is itself largely AI-co-developed, and the serializable format is a big part of why that works.
- **Auditability.** Framework rules (token usage, no raw DOM, component naming) are mechanically checkable against the compiled tree — the `.frank-audit` snapshots in git history show this running continuously.

No mainstream stack has an equivalent. A React app is not a document; it is a build graph. You cannot hand "the app" to an editor, an auditor, or a model as one structured object.

6 · Honest Comparison vs. a Conventional Stack

The fair baseline is React/Next.js with an npm-workspace design-system package — the setup most teams would use for six sibling apps.

Dimension	Symbols (this workspace)	React/Next + npm workspace
Cross-app component reuse	WIN Bare-key resolution through merged context. Linking a sibling library is two config lines; no publishing, no version ceremony for local work. Override = define locally, merger does the rest.	Achievable, but requires a maintained package boundary: exports map, build step, semver discipline, and explicit imports everywhere. Per-app overrides need wrapper components or prop drilling.
Design-system enforcement	WIN Tokens are the styling syntax itself; themes are semantic and carry dark/light variants. Rebrand ≈ 300 lines. Violations are mechanically auditable against the compiled JSON.	Tokens live beside the code (Tailwind config, CSS vars, theme object). Enforcement is social + lint rules; raw values leak constantly in practice.
Code volume	WIN Declarative object trees with no JSX/component boilerplate, no import blocks, no className plumbing. A 19-page app fits in	Equivalent apps typically carry their own component tree per app or a heavyweight shared package;

	~14 K lines including logic; consumer apps carry 7–8 components.	boilerplate (imports, props interfaces, wrappers) adds real volume.
Serializable context / AI & tooling leverage	WIN Whole app = one JSON document (~620 KB measured). Editor, runtime, audit tooling, and AI agents share one representation.	No equivalent. The app exists only as a build graph; tooling and AI work file-by-file.
Runtime/bundle character	Direct DOM rendering from object trees, no virtual-DOM diff layer, no JSX compile target. Architecturally lean — though this report ran no benchmarks, so no performance numbers are claimed here.	Mature, heavily optimized, but carries the React runtime and reconciliation model as a baseline cost.
Ecosystem & hiring	COST Small community, framework-specific knowledge, fewer Stack Overflow answers, fewer ready-made integrations. New developers must learn DOMQL conventions and the frank constraints (no closures, serializable functions).	Enormous ecosystem, every library has a React binding, hiring pool is the industry default.
Tooling maturity	COST The compile step (`frank`), the two-file shared-library config, and cache-clearing rituals after framework changes are real operational overhead. The CLI guards against drift, but the guardrails exist because drift happens.	Decades of editor support, type checking, debugging, and CI patterns out of the box.
Type safety	COST Plain objects with string-key references resolve at runtime; a typo'd component name fails silently as an empty element rather than at compile time.	TypeScript catches broken references, prop mismatches, and refactors statically.

7 · The Honest Part: Where This Workspace Falls Short of Its Own Ideals

A review that only praised would be marketing. These findings are measured from the same repository and are worth stating plainly, because they show the difference between what the framework *enables* and what discipline still has to *maintain*:

7.1 Navigation components were copied, and the copies drifted

Each app carries a local `TopBar`, `BottomBar`, `AdminSidebar`, and `ThemeToggle`. Checksums show five apps holding **five different versions of BottomBar** — not five brand variations of a shared base, but five diverged full copies. The chatbot library ships canonical versions of all of them; the apps shadow them with local rewrites instead of extending them. The framework's override pattern (`extends: 'BottomBar'` + a handful of prop overrides) exists precisely to prevent this and is not being used here.

7.2 Shared page templates exist but have zero consumers

The chatbot library exports five `Generic*` full-page templates (`GenericChatPage`, `GenericFaqPage`, `GenericAdminPage`…). Measured consumption across all five consumer apps: **zero** `extends: 'Generic…'` references. Every app still carries its own near-identical `chat.js` / `faq.js` (benefits' and hotcard's chat pages are visibly the same code, down to the same `hc-dot-anim` style id). The mass of `*.legacy.js` pages across apps shows a migration in progress — honest status: the extraction happened, the adoption hasn't.

7.3 Token rules are violated where nobody looked

Despite the "no raw values" rule, benefits' seven local components contain **76 raw px/hex occurrences**, and several pages inject keyframe CSS via `document.createElement('style')` in `onRender` — exactly the raw-DOM escape hatch the project rules forbid. The framework can express all of this properly (animation tokens exist in the design system); the shortcuts were taken anyway.

7.4 uhy was cloned, not derived

`uhy` shares byte-identical component files with hotcard (matching md5s for ThemeToggle and AdminSidebar). Clone-and-tweak got the app shipped fast — which is itself a kind of reuse — but it is the npm-era failure mode reproduced inside a framework designed to avoid it. The right shape would have been a shared hotcard-family library, or deeper use of the chatbot tier.

> **The honest synthesis:** none of these findings are framework failures — every one has a first-class mechanism that would have prevented it (override pattern, Generic pages, animation tokens, shared libraries). They are discipline failures that the framework makes *visible and mechanically fixable*: because everything compiles to one auditable JSON tree, this report could measure the drift in minutes. In a conventional stack, the same rot would exist and be far harder to even detect. But the

> gap between "enabled" and "achieved" is real, and closing it is ongoing work — the legacy-page migration is the evidence it's being closed.

8 · Verdict

The reusability claims hold up under measurement. Five branded applications run on a 7–8 component local footprint each, against a 157-component shared surface. Rebranding costs ~300 lines. New full apps shipped within two days of the shared library existing. The entire application state is one auditable, AI-consumable JSON document — a property no mainstream stack offers.

The costs are equally real: a small ecosystem, framework-specific conventions to learn, a compile step to respect, runtime (not compile-time) reference resolution, and — as sections 7.1–7.4 show — leverage that only pays out when the override-don't-copy discipline is actually followed.

For the specific shape of work this workspace does — many sibling products sharing one domain core, heavy AI co-development, a design system as the product's backbone — Symbols delivers leverage that would take a conventional stack a dedicated platform team to replicate. The strongest evidence isn't any single feature: it's that `dashboard`, the one app built without the shared domain tier, needed **57 local components** to do what every other app does with **7**.

All figures measured directly from the repository on 11 June 2026: component/page counts via directory listings, LOC via `wc -l` (excluding node_modules, audit snapshots, lib caches), drift via md5 checksums, consumption via grep over app sources, timeline via git first-commit dates, compiled size via `.symbols_local/project.json`. No performance benchmarks were run; no performance numbers are claimed.