Technical Report defuss · October 2026

Verified Agentic Engineering in Practice

Agent-authored control programs and persistent adaptation across a growing component system

Aron Homberg[1]

[1]independent researcher - defuss open-source project

defuss-vae | Code | ARCH.md | Slides | Issues

Data The case-study measurements are frozen at the study cutoff, October 7, 2026, 17:58 UTC, shortly before defuss-shadcn release 0.9.7. They cover 33 calendar days, from September 5 through October 7, 2026. The four engineering changes are measured from the session transcript of October 6. The general controller implementation is described from defuss-vae 0.8.0 of October 7; its later features are distinguished from those used during the study.

Abstract

Modern Agentic Engineering approaches still face challenges with inconsistencies in task execution completeness and consistency. This phenomenon specifically affects large-scale change requests as well as new feature requests which leads to an increase of work left for human reviewers. Verified Agentic Engineering (VAE) is a working method for such requests, not a new model, agent or UI framework. Its implementation, defuss-vae, adds skills, hooks and Python programs to an agent harness: the program that runs a model and its tools, such as Claude Code, Codex, and alike. The agent writes and revises programs that check its work, select the next unit of work and return instructions through the harness. A verifier reports unmet acceptance criteria; a document controller presents bounded passages for review. The routines shown here require no AI model call. The harness passes their output to the model, which edits the artifact and invokes the next step within human-defined policy. Across tasks, the agent can retain findings as checks, instructions, reusable tools or bounded prompt memory. VAE-DIALECT provides compact notation for evidence and scope; stack templates provide toolchain commands and conventions.

This report describes the method's development in defuss-shadcn, a UI component system built from semantic HTML, CSS and JavaScript, from September 5 to October 7, 2026, under one maintainer, Aron Homberg, using Qwen3.8-Flash-Next, Claude Opus 5.5 and Claude Fable 5.1. The repository grew from 55 to 231 components across 220 commits and 17 releases. In one session, the agent added typed API documentation for all 59 interactive components (241.9 minutes, one human correction), raised reported line coverage from 31.6% to 75.6% (20.6 minutes), moved 228 component folders into a new layout (43.4 minutes), and removed 1,147 type errors (27.3 minutes to the final passing gates). All 85 minified artifacts compared for the type-error change were byte-identical. These observations document consistent repository-wide changes under the programmed self-feedback-loop regime of the defuss-vae method; the study does not isolate the method's effect on correctness, cost or human review effort.

1 Contribution and scope

Modern Agentic Engineering approaches still execute tasks incompletely or inconsistently. Large-scale change requests and new feature requests are affected most: the agent has to update every affected module and check that the new requirement holds across the resulting system, and each missed module or inconsistent implementation adds to the work left for human reviewers. Verified Agentic Engineering (VAE) addresses this problem with a programmed self-feedback loop: the agent writes and revises the programs that check its work and direct further action. The intended trade-off is additional automated work in exchange for more complete and consistent changes and less manual review.

The same programs can support later changes by retaining checks for the requirement. The interface also extends to tasks that need organized review: a controller can present one document section or data partition at a time, together with the applicable instructions.

VAE combines two feedback loops. Within a task, program output identifies a failed condition or the next work unit. Across tasks, the agent retains findings as checks, instructions and selected memory. Explicit module contracts provide stable conditions to evaluate and boundaries for dividing work among agents. VAE-DIALECT marks evidence, hypotheses and unknowns in compact notation. Stack templates provide reusable toolchain procedures.

This report describes the implemented method, four repository-wide tasks and the procedure changes prompted by their findings. It covers one component system, one maintainer, models from the Qwen and Claude families, and 33 calendar days. The observations concern the combined system; they do not separate the contributions of the model, harness, verifier, instructions, dialect, templates and memory.

The feedback loop has precedents. ReAct interleaves reasoning, actions and observations (Yao et al., 2023); SWE-agent studies how interfaces for editing, navigation and execution affect software-engineering agents (Yang et al., 2024). VAE's emphasis here is the use of agent-authored control programs and retained project procedures during continuing repository development. Section 7 relates this adaptation to prior work on agent memory and reusable skills.

2 Programmable feedback and persistent adaptation

VAE combines a language model, an agent harness and programs the agent may create or adapt within its authorized scope. The human sets the objective and policy. Programs encode selected requirements and return findings or instructions from the current work state. The harness passes those outputs to the model, which interprets them, acts and invokes the next step (Figure 1).

Autoregressive generation occurs within each model response; the harness and programs provide the outer loop across calls. Intermediate corrections can proceed without a new human prompt at every iteration. The model still has to interpret the evidence, edit the artifact and, when needed, improve the tools.

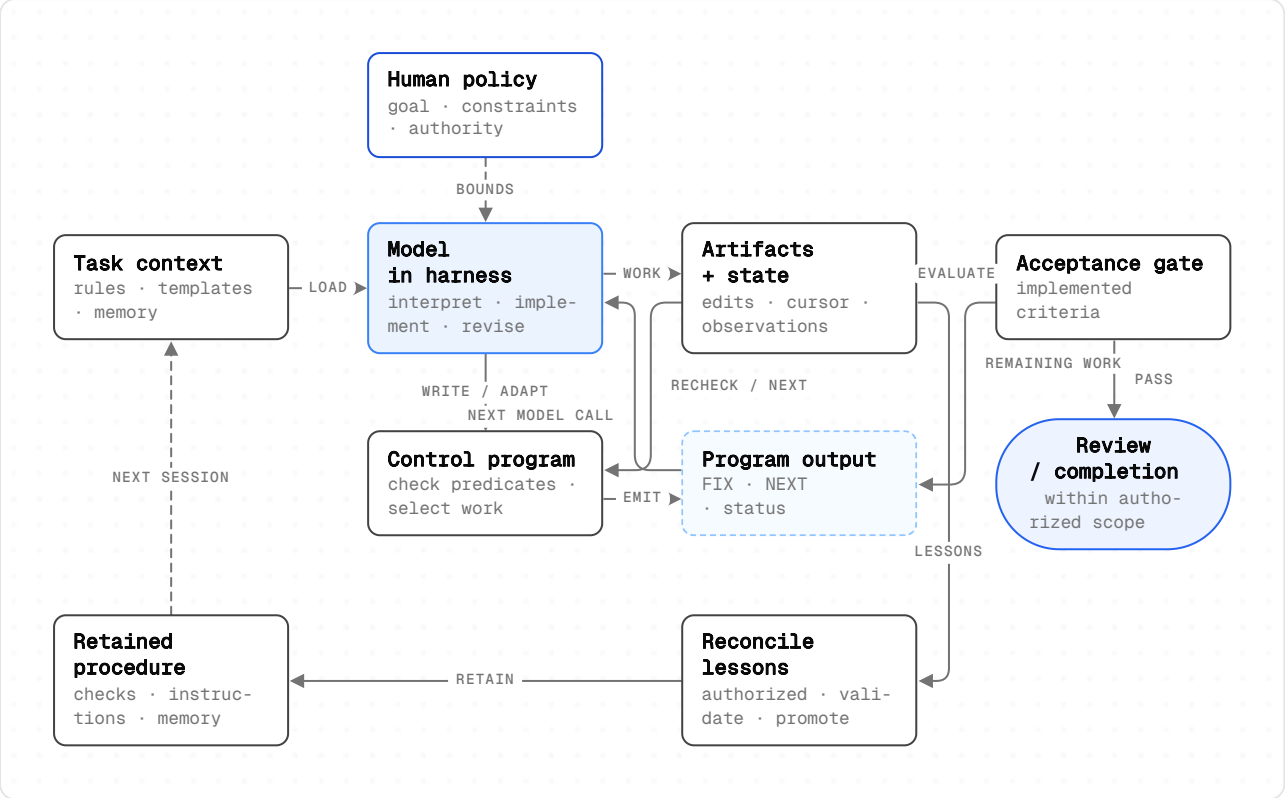


Figure 1. The general VAE mechanism. Programs evaluate checks or select bounded work and return instructions to the model through its harness. The agent changes the artifact, invokes the next step and can revise the task-specific programs. A second loop reconciles findings into retained checks, instructions and memory for later sessions. Human policy sets the objective, acceptance criteria and adaptation authority. The diagram describes the method; section 2.3 states the enforcement and initiation boundaries of defuss-vae 0.8.0.

2.1 General feedback protocol

A controller receives an artifact or task state and returns a finding or work instruction. It may evaluate a predicate, select the next work unit, supply context and rules, or name a command. The agent then changes the artifact or task state. If a finding exposes a missing check, the agent may extend the controller and validate the extension against the governing requirement.

The interface is ordinary tool output. For example, a build script invokes a verifier, whose standard output returns findings and repair instructions through the harness. Different controllers provide different evidence: a predicate check evaluates its specified condition, while a document controller selects passages and review rules. Semantic judgments about those passages remain with the model.

Controller	Program operation	Output returned to the harness	Model work	Evidence in this report
Code verifier	Evaluate executable acceptance criteria	Failing condition, affected location and repair instruction	Diagnose and repair the artifact; extend checks when justified	Repository case study and controlled output capture
Document window	Select a text part, preceding context and review rules	Bounded text, applicable instructions and next command	Evaluate meaning, edit the part and continue	Source-inspected controller and controlled output capture
Spreadsheet partition	Select a row group or sheet region and applicable task rules	Partition, task instructions and next partition	Apply the requested transformation and inspect exceptions	Architectural illustration

Table 1. Three uses of the controller interface. This report inspects code checks and document windows; the spreadsheet partition is an architectural illustration.

2.2 Program output and the model boundary

The harness carries outputs between the agent and its control programs. In defuss-shadcn, the coding agent wrote and extended the project verifier in response to new requirements (section 4.4) and retained findings (section 2.4). Figure 2 presents the exchange as a conversation, using small fixtures to show a built-in defuss-vae rule and the document controller's output verbatim.

The outputs below were captured from unmodified defuss-vae 0.8.0 functions on controlled fixtures. The verifier example places the temporary probe marker, which the hygiene rule searches for, on line 2 of `component.ts`. Ordinary code detects the marker and formats the failure. After the marker is removed, the same check reports success.

The document controller uses a different operation. `vae.py prose --walk` selects a bounded part of a page and prints its preceding context, static findings and next command. The first part includes the full review catalog; subsequent parts identify the rules. The catalog has 58 rules covering evidence, logic, terms, relevance, structure, language and typography. The document controller organizes the review, while the model evaluates the text and makes the necessary edits.



Figure 2. The agent and its control programs as a conversation. The verifier and document-window messages are captured output from small fixtures using defuss-vae 0.8.0, quoted verbatim; the walk excerpt omits the long CHECK EVERY `rule` index line between STATIC and DO. The agent's messages state the action each output asks for; they are not captured text. The shown predicate evaluation, window selection and output formatting make no AI model call. In harness-driven operation, the harness supplies the output to the model's next invocation, where interpretation and the requested editing occur. These captures demonstrate the program interface, not the original production session.

The project's own verifier follows the same pattern with its own wire format: each failing check prints a `fix:` line naming the command or file to change. When a page figure disagreed with its measurement on October 7, the `stat figures` check named `bun scripts/stat-figures.ts` and a documentation rebuild, and running them resolved the failure.

The document controller's final `VERIFIED[walk]` output depends on the requested final part and static findings. The doc and `doc-edit` skills prescribe the review procedure, but the gate does not validate traversal. A review attes-

tation records the agent's claim that it reviewed the artifact.

The source of feedback matters. Self-Refine uses an LLM to generate, critique and revise an answer (Madaan et al., 2023). In the examples here, program code generates the predicate result or review window; the model performs the diagnosis or semantic review. A deterministic controller therefore does not make every judgment in the workflow deterministic.

2.3 Harness enforcement and adaptation authority

defuss-vae implements the VAE method as skills, hooks and Python programs. At session start a hook supplies rules, selected stack defaults, repository memory and open leads. The gate runs verification and checks review and documentation attestations for the relevant work state. defuss-shadcn configures its own commands and deterministic rules in `.agents/VERIFY.py`, on top of the 87 custom static verification checks implemented in `scripts/verify.ts`, which `VERIFY.py` runs as one of its commands. Missing or indeterminate required evidence blocks the monitored commit action.

With the commit hook active, a monitored commit requires a passing gate for the relevant work state. A failed gate supplies instructions for further work. This enforces the configured commit boundary; it does not ensure that every task runs uninterrupted or reaches completion.

Adaptation is subject to the same policy as the task. The agent may add checks, revise procedures and create tools within its authorization, while acceptance criteria remain those of the request and project policy. Weakening a check does not satisfy the original requirement. When a person invokes `wrap`, the skill prescribes reconciliation of retained lessons and removal or revision of obsolete memory. Within the authorized scope, the agent can carry out intermediate corrections without a separate human prompt for each one.

2.4 Reconciliation and bounded prompt memory

Failures and review findings become episodes with supporting evidence. During reconciliation, the agent compares those episodes with the current code, tests and retained rules. A durable lesson becomes an executable test or verifier rule where that form fits. Otherwise it remains a scoped instruction, memory entry or working command. Later sessions inherit this state and can apply the retained procedure without reconstructing the original failure.

Here, self-learning means changes to the working system's stored evidence, tools and procedures. It does not involve updating model weights. The agent can improve the artifact and the programs that guide or evaluate later work. A retained change needs evidence for its scope. Any current human instruction outranks an older memory entry.

The injected memory is bounded. In defuss-shadcn, the header of `.agents/MEMORY.md` sets a budget of 4 KiB and that of the command gist 2 KiB; defuss-vae 0.8.0 adds a limit of 240 characters per memory line, and `vae.py doctor` checks all three. Unresolved leads are retained for reconciliation. When an executable check supersedes a narrative lesson, that narrative can be removed with a reference to the check. Lessons can therefore move from repeatedly injected prose into verifier programs, tools and tests. The total repository of checks and procedures may grow even while prompt memory remains small. In defuss-shadcn, `MEMORY.md` held eight entries in 2,311 bytes from October 6 (release 0.9.6) through the end of the study.

Retention is useful when a later task can apply the lesson within its supported scope. Whether that prevents repeat failures or reduces effort requires measurement; storing a lesson alone establishes neither.

2.5 Dialect and stack templates

VAE-DIALECT combines natural language with compact operators and evidence labels: `VERIFIED[scope]` marks a directly supported state-

ment, `HYPOTHESIS[scope]` marks an inference to test, and `UNKNOWN[scope]` marks a material gap in the evidence. Operators express conditions, precedence and reasons. Each statement retains its scope and supporting evidence; correctness takes precedence over compression.

The notation is intended to make unsupported assertions and ambiguous instructions easier to identify by requiring an explicit epistemic status. Compact forms may also reduce repeated wording; their effects on token use are discussed in section 4.8. A `VERIFIED` label is meaningful only within the scope of its supporting evidence.

Stack templates provide toolchain procedures before the agent begins work. The controller selects defaults for the repository's detected stacks and inserts them into session context or the managed instruction block. When a required command is missing, the verifier includes the corresponding default in its repair instruction. Figure 3 shows the defaults selected for a Python project.

```
python lint: `uv run ruff check .`
python test: `uv run pytest`
python coverage: `uv run pytest --cov`
```

Figure 3. The stack-template defaults selected for a Python project, as the controller inserts them into the session context or the managed instruction block.

Selection and formatting require no AI model call. These strings supply guidance; the gate executes the repository's configured commands. Templates are intended to reduce repeated tool selection, setup exploration and unnecessary specification work. Along with the dialect and retained procedures, they offer a way to spend more of the context on the current task. However, this report does not isolate their effects on unsupported claims, task errors, token use or elapsed time.

Templates cover Go, Rust, the JVM, .NET, JavaScript and TypeScript, Python, and web projects. The author selected these as practical defaults based on engineering experience. A repository retains its existing toolchain unless a person approves a migration; `.agents/VERIFY.py` can configure different lint, test, coverage and end-to-end commands.

The templates also define procedures for planning, implementation, review, documentation, subagent coordination and long-running work. A long job runs detached with a PID file and a log whose lines carry ISO 8601 timestamps. The gate records failures and findings with their session identifiers, and each attestation names the code fingerprint it covers.

`vae.py init` scaffolds a project layout with `Makefile` targets for `setup`, `start`, `stop`, `status`, `log`, `metrics`, bench, `test`, `coverage`, `lint`, e2e and `verify`. The service actions and status codes draw on the LSB Core Specification 5.0, §22.2; the additional targets and repository paths are project conventions. Services write `var/log/<service>.stdout` and `.stderr` and keep their PID in `tmp/<service>.pid`. Programs read `input/` and write `output/`, separating runtime state from source code.

Sub-agents use separate git worktrees and a shared registry, `.agents/SWARM_STATUS.yaml`. Registry writes use a lock, atomic replacement and a read-back check. Only the owning process may change its entry; overlapping path claims are rejected, and liveness is checked against the process table. These rules coordinate agents that use the registry protocol. They do not establish that all concurrent repository activity is free of races.

3 Case study and evidence

defuss-shadcn is a component system built from semantic HTML, CSS and JavaScript, with TypeScript sources and no framework or build step required by consumers. It was forked from shadcn-html with 55 component folders. The study follows development from September 5 to October 7, 2026, under one maintainer using models from the Qwen and Claude families.

Each interactive component has an explicit contract: named states, a store, a typed API and a `render()` operation checked against its authored markup. This use of stated obligations and assertions follows the design-by-contract tradition (Meyer, 1992). Shared behavior, including DOM access and component-directory discovery, has an identified owner. These conventions allow common checks to run across components and shared changes to reach their callers. The four tasks below use this structure at repository scale.

The project had an instruction file and its own verifier from the first commit. defuss-vae was first published on October 1 and wrapped the project verifier from October 6.

The session ran in Claude Code. Every model response recorded inside the four task windows of section 4 came from `claude-opus-5-5`; later sessions on October 7 also used `claude-fable-5-1`, a model of the same family.

In September, Qwen3.8-Flash-Next was used while the component count grew from 55 to 77.

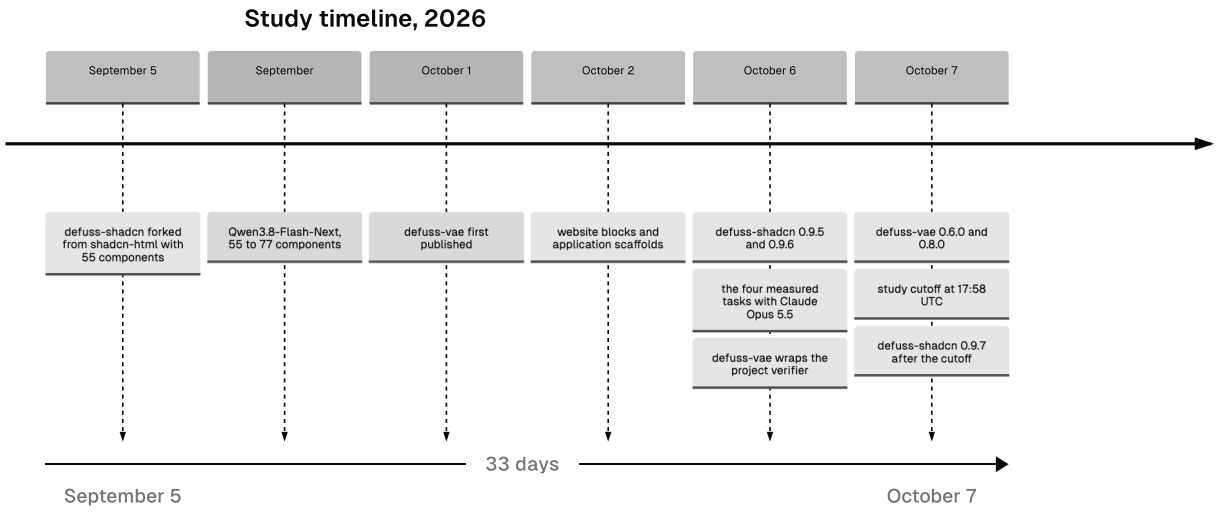


Figure 4. The study period, 33 days from the fork on September 5 to the cutoff on October 7, 2026: the models used, the defuss-vae and defuss-shadcn releases and the day of the four measured tasks. The columns are in order, not to scale.

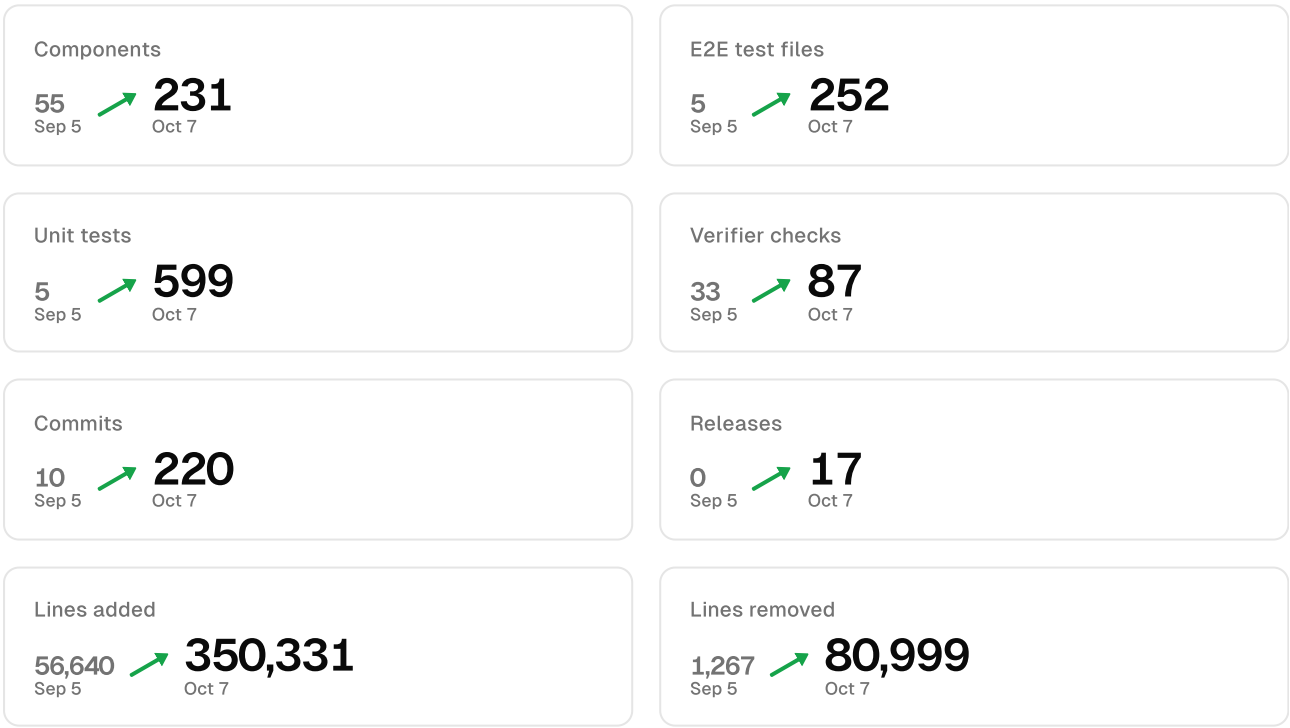
4 Results

The October 6 session contains four repository-wide tasks: typed API documentation, shared-contract tests, component layout and type-error correction. For each, the agent applied a requirement across the relevant components and updated checks or supporting tools where the new requirement needed them.

Task	Reported endpoint	Minutes	Output tokens	Tool calls	Human corrections
Typed API documentation	Final green gate, all 59 interactive components	241.9	347,614	437	1
Test coverage	Final reported coverage measurement	20.6	39,926	48	0
Repository layout	Final green gate, all 228 component folders	43.4	76,358	98	0
Type errors	Zero diagnostics and final green gates	27.3	82,855	130	0

Table 2. Four changes from one session, measured from each request to its reported endpoint. API documentation, layout and type correction end at final green gates; coverage ends at its final reported measurement. The windows include contemporaneous work: section bundles in the first, and a CI workflow and report edit in the second. Times and token counts describe those task windows, rather than isolated controlled trials. Human corrections count correction prompts within each window, not all human input or oversight.

4.1 Growth and verification coverage



The repository grew from 55 components at the fork to 231 at the study cutoff on October 7. The boxes compare the last commit of September 5 with the cutoff commit; the line counts are cumulative from the fork, whose import they include. At the end of the first day the repository held 5 E2E test files and one unit-test file with 5 tests; at the study endpoint, 252 E2E files and 35 unit-test files, whose last full recorded run held 599 tests. Over the same span the verifier grew from 33 checks to 87, counted as the distinct labels of the `check( ... )` calls in `scripts/verify.ts`. Its recorded October 5 run, on an earlier tree, passed 78 gate instances, the result lines the verifier prints; warnings are not counted as passes.

The interval contains 220 commits and 17 releases. The largest component-count increase followed the addition of website blocks and application scaffolds on October 2. These counts establish the size and activity of the project during the study; they do not independently measure correctness or the method's causal effect.

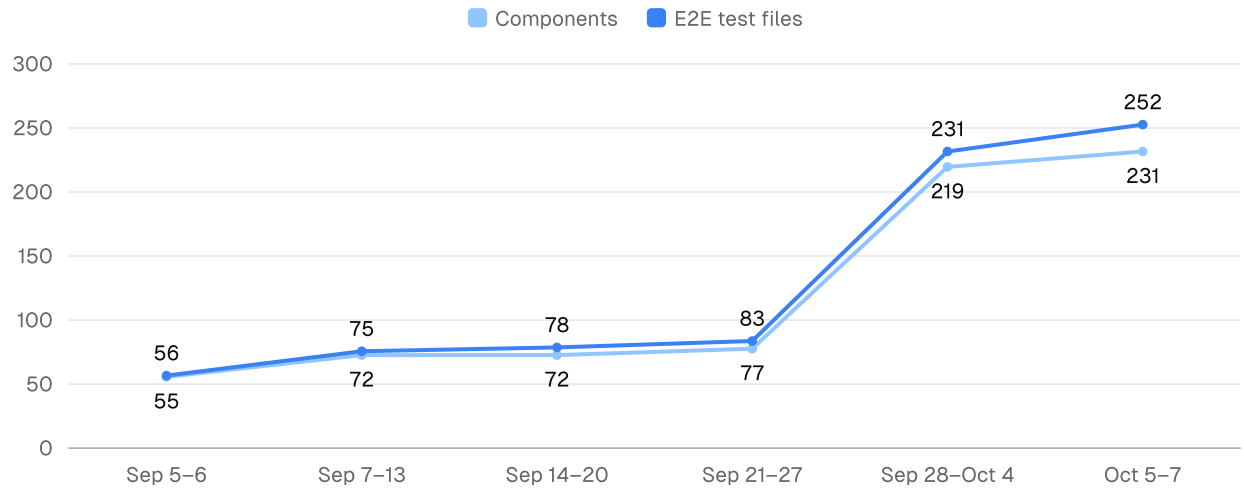


Figure 5. Components and E2E test files at the last recorded commit of each calendar week, read from git objects. The first and last intervals are partial. The fixed endpoint is October 7, 2026: 231 components and 252 E2E files. The fifth interval includes the website blocks and application scaffolds added on October 2. The weekly grouping uses the original measurement's UTC+2 convention; task timestamps elsewhere are UTC.

4.2 Typed API documentation

The first requirement was to add typed API documentation for all 59 interactive components. Every member needed a TypeScript signature, every argument a description, every event typed detail, and every state its accepted configuration. The shared State API also had to be typed for each component's own states and checked in both source and rendered documentation.

The final gate passed 241.9 minutes after the request, including one hour of hibernation due to low battery. The task window used 347,614 output tokens and 437 tool calls, with one human correction. All 59 interactive component sources changed. The source verifier checked all 59 components, and the E2E audit opened all 59 rendered API pages. The first passing result had missed configuration information; section 5.1 describes how that finding expanded the verification scope.

The requirement arrives
The request arrives during another task; the work proceeds together.
05:27 UTC

First green gate
Verify, review and documentation attestations.
06:59 UTC

One human correction
The maintainer opens a second page: the configuration each state accepts is missing. The maintainer clarifies the requirement.
07:02 UTC

Final green gate
All 59 interactive components complete, with an E2E audit of all 59 rendered API pages.
09:29 UTC

Figure 6. API-documentation timeline on October 6, 2026, read from the session transcript. The first green gate was followed by one human correction; the final gate included an audit of all 59 rendered API pages.

The change added 3,455 and removed 408 lines across the component sources. Across all authored files in the window, 246 files changed, with 7,570 additions and 1,537 deletions; that larger total

includes section-bundle work and a documentation pass. The compiler ratchet fell from 1,212 diagnostics to 1,147. These are separate scopes: the documentation requirement covered the 59 interactive components, not every component in the project.

4.3 Test coverage

The next task raised reported browser-mode Vitest/V8 line coverage from 31.6% to 75.6% in 20.6 minutes, using 39,926 output tokens and 48 tool calls. Recorded covered lines increased from 5,460 to 15,880. The work added 147 lines of tests and an 11-line component fix. The measurements include loaded documentation code and third-party packages; the percentages are those of their respective instrumented runs.

The agent reused 63 existing interactive fixtures: the 59 component fixtures and four compositions. The first contract test exercised every declared state and checked agreement among `getState()`, the store and `render()`, while requiring an undeclared state to throw. It exposed a missing `settled()` operation in the cookie-consent controller. The second test operated controls and fields and checked for thrown exceptions and unhandled promise rejections.

Measure	Before	After
Lines, step 1: states	31.6%	↗ 66.9%
Lines, step 2: controls	66.9%	↗ 75.6%
Statements, step 1: states	-	61.0%
Statements, step 2: controls	61.0%	↗ 71.8%
Functions, step 1: states	-	60.0%
Functions, step 2: controls	60.0%	↗ 71.1%
Branches, step 1: states	-	38.6%
Branches, step 2: controls	38.6%	↗ 50.5%
Unit tests, step 1: states	420	↗ 497
Unit tests, step 2: controls	497	↗ 564
Coverage floor the gate enforces, step 1: states	30%	↗ 60%
Coverage floor the gate enforces, step 2: controls	**60%**	↗ **70%**

Table 3. Coverage measured with Vitest/V8 in browser mode before the task and after each of its two steps, each row comparing two measurements; step 1 was measured at 13:12:56 UTC and step 2 at 13:29:08 UTC. Line coverage rose from 31.6% to 66.9% and then 75.6%; the enforced floor rose from 30% to 60% and then 70%. The unit-test count rose from 420 to 497 and then 564. Unrecorded baseline metrics are marked "-". Each percentage belongs to its recorded instrumented population.

The shared fixtures let two tests exercise all interactive components. Their assertions check the stated contracts and error paths; line coverage records which instrumented lines executed. In a study of five Java systems, Inozemtseva and Holmes found only a low-to-moderate association between coverage and mutation-based test effectiveness after controlling for suite size (Inozemtseva and Holmes, 2014). Coverage here should likewise be read as execution coverage, alongside the specific assertions and the defect they exposed.

4.4 Repository layout

The third request was to make the source tree follow the documentation sidebar. Every component needed a `section:` entry in its skill, a matching section directory and a stable published path. The agent moved all 228 component folders into 19 sections in 43.4 minutes, using 76,358 output tokens and 98 tool calls, without a human correction.

The change centralized directory discovery and updated 11 tools to use the shared module. It affected 611 authored files, with 1,008 additions and 625 deletions after rename detection. The compiler ratchet remained at 1,147 diagnostics; 570 unit tests and 246 E2E files passed at completion. The agent corrected intermediate failures in path discovery, generated links and configuration before the final gate.

Measure	Count before	Count after
Section folders under `src/components/`	0	↗ 19
Component folders inside a section folder (moved with `git mv`, history kept)	0	↗ 228
Files in the component folders (all moved, none added or removed)	582	582
Skills declaring `section:`	0	↗ 228
Sibling links routed through the section	0	↗ 261
Component sources importing from the section depth	0	↗ 59
Tooling readers using the shared directory owner (`component-dirs.ts`)	0	↗ 11
Documentation components rewired	-	4
Tests rewired	-	8
New modules	-	3
New test files	-	1
Authored files changed (renames detected)	-	611
Lines added	-	+1,008
Lines removed	-	−625
Type diagnostics (the compiler ratchet)	**1,147**	**1,147**

Table 4. The repository-layout change, from git and verifier output. All 228 folders, containing 582 files, moved under 19 sections. Published component paths remained flat. The recorded minified sizes were unchanged; this measurement alone does not establish byte identity. An arrow marks the direction of a change; "-" marks a count that has no before value.



Figure 7. Three of the 228 component folders before and after the move (October 6, released in 0.9.7): each folder keeps its files and moves into the section that lists it in the sidebar. On screen, the divider shows more of either tree when dragged.

The new layout check covers three violations: a missing `section:`, a component in the wrong section and a component outside all sections. Each fails with an instruction identifying the move or front-matter correction. The change thus updated both the repository layout and the checks applied to later work.

4.5 Type errors and output equivalence

Later in the same session, the agent was asked to remove the remaining 1,147 compiler diagnostics without changing the shipped output. It reached zero diagnostics in 12.5 minutes, using 71,958 output tokens and 100 tool calls. The final green gates were reached after 27.3 minutes, 82,855 output tokens and 130 calls, with no human correction.

A type change in the shared query module removed 707 diagnostics before any component was edited; the other 440 required local corrections. The complete change touched 45 component sources and three shared or type files, adding 276 lines and removing 210. At completion, 571 unit tests and 246 E2E files passed, and all 85 compared minified artifacts were byte-identical to the preceding build.

Measure	Count before	Count after
Type diagnostics in the component program	1,147	✓ 0
… after the type fix in the shared query module	1,147	✓ 440
… after the local corrections	440	✓ 0
Component files with diagnostics	52	✓ 0
Component sources changed	-	45
Shared modules and type declarations changed	-	3
Source lines added	-	+276
Source lines removed	-	−210
Source-reading tools that accept a type argument	0	↗ 3
Shipped minified artifacts, byte-identical to the previous build	**85**	✓ **85**

Table 5. Type-correction results from git, compiler output and byte comparison. Equality applies to the 85 minified artifacts compared before and after this change. The readable JavaScript differed in two lines because an erased cast left parentheses that the minifier removed. A check marks a result the change aimed for: fewer diagnostics, identical bytes. An arrow marks the direction of any other change; "-" marks a count that has no before value.

4.6 Human review effort

On September 7 the maintainer reviewed the documentation site for about eight hours and filed 52 issues, including missing examples, field hints, dead links and visual concerns. Eighteen issues were closed within twelve hours. The remainder were closed in a batch on September 28, so those closure timestamps are upper bounds on the corresponding fix dates.

4.7 Changes to committed code

Across the study interval, git recorded 350,331 additions and 80,999 deletions in authored files, excluding generated output. The first day's 56,640 additions include the imported fork. Documentation accounts for 69,338 deletions, including the migration from static pages to MDX.

Across components, tests, the shared runtime and theme, git recorded 8,079 deleted lines and 165,668 added lines, a deletion-to-addition ratio of **4.9%**. These aggregate counts do not measure the survival of individual lines. They describe committed change volume, including migrations, rather than a defect or rework rate.

Area	Added	Deleted	Net	Deleted / added
Components (src/components)	+99,100	−5,793	+93,307	5.8%
Documentation (src/documentation)	+166,874	−69,338	+97,536	41.6%
Tests (tests/)	+57,578	−2,029	+55,549	3.5%
Tooling and verifier (scripts/)	+10,726	−1,910	+8,816	17.8%
Theme and tokens (src/theme)	+5,786	−119	+5,667	2.1%
Shared runtime (src/shared, src/core, src/types)	+3,204	−138	+3,066	4.3%
Other (instructions, skill templates, configuration)	+7,063	−1,672	+5,391	23.7%
All authored files	+350,331	−80,999	+269,332	**23.1%**

Table 6. Added and deleted line events by area, from `git log --numstat` over September 5 through October 7, with rename detection and generated files excluded. The initial fork import is included. The ratio is deletions divided by additions.

4.8 Execution and context cost

The recorded October 5 run of the project verifier evaluated 78 gate instances in 10.2 seconds on an Apple M4 MacBook Air. That is the verifier command's duration, not the complete build, screenshot and test pipeline. Task-window elapsed times and output-token counts are reported separately in Table 2.

The session that produced the first version of this report also records input-token accounting. Across its requests, 580,660,897 input tokens were cache reads, 2,609,778 were cache writes and 2,330 were uncached input. Cache reads account for 99.6% of the sum. These are accumulated usage counters, including repeated reads of cached context; they are not the size of a single context or a count of unique text.

The three categories are priced very differently. At the Claude API list prices for Opus 5.5 at the time of writing (Anthropic, 2026), a cache read costs $0.20 per million tokens, 95% below uncached input at $4 per million. A cache write costs more than uncached input: $5 per million for the default 5-minute cache and $8 for the 1-hour cache, which every cached prefix of this session used. At these prices the recorded input comes to $137.02: cache reads were 99.6% of the tokens and 84.8% of the cost, cache writes 0.4% of the tokens and 15.2% of the cost. The same tokens at the uncached price would cost $2,333.09. These are list-price figures for the recorded counters, not the amount the session was billed.

Input usage category	Price per million	Tokens	Cost
Cache reads: least expensive, 95% below uncached input	$0.20	580,660,897	$116.13
Cache writes, 1-hour cache: most expensive, twice uncached input	$8.00	2,609,778	$20.88
Uncached input: the base price	$4.00	2,330	$0.01
All input	-	**583,273,005**	**$137.02**

Table 7. Input-token accounting for the session that wrote the first report version, through the recorded October 7 measurement (04:20 UTC), priced at the Claude API list prices for Opus 5.5 read on October 8, 2026 (US dollars). Every cached prefix of the session went to the 1-hour cache. The first request recorded 69,980 cache-write tokens. These counters include the session context and cannot be attributed exclusively to the instruction file. The costs are list prices applied to the counters, not the amount billed. These prices apply to token-based (API) billing only. A monthly subscription charges a flat fee within usage limits instead, so for an individual developer on such a plan the per-token prices matter only through those limits. The maintainer reports working on the Claude Max plan of 2026 and, across this and several other projects, never exceeding 90% of its weekly limit.

Caching, compact notation and reusable procedures affect different parts of the cost. The counters show how input was served; they do not measure savings from VAE-DIALECT or templates. Working templates may reduce failed setup attempts, and compact notation may shorten instructions. Shorter text does not necessarily mean fewer tokens: that depends on the tokenizer. The net effect on token use and elapsed time remains unmeasured. The comparisons proposed in section 6 would test those effects separately.

5 Adapting checks, instructions and memory

The following cases illustrate four responses to findings: extending a check, retaining an instruction, obtaining a human acceptance decision and reconciling memory. They explain how the procedure changed; they are not a sample from which to estimate failure rates or the effectiveness of each response.

5.1 Extending the verifier

The typed API documentation first passed its gate at 06:59 on October 6. At 07:02 the maintainer opened a second page and found that the configuration accepted by each state was missing. The check had covered source information without establishing that every rendered page exposed it.

Under the VAE method, the agent corrected the documentation and extended the work to an E2E audit of all 59 interactive-component pages (`tests/e2e/api-docs.e2e.ts`, added in release 0.9.6). The final green gate was recorded at 09:29. The human supplied the missing acceptance observation; the agent supplied the repair and the broader check. The correction therefore changed both the artifact and the verification scope. Once an executable check embodies that requirement, later work can test it without reconstructing this exchange.

5.2 Retaining lessons in instructions

Working on the project, the agent recorded failures at module boundaries: the dialog script claimed a dialog owned by another component, and the tabs script claimed the BibTeX component's tab list. At 03:56 UTC on October 7 the maintainer asked to design a guideline for component ownership as a strong recommendation in `AGENTS.md`; the agent drafted it, the maintainer shortened the wording at 04:19, and a commit added it at 04:38. The instruction states that one module owns each concern and other modules use that owner. The same rule

entered the `plan` and `implement` skills of defuss-vae 0.6.0 at 08:49 that day.

This sequence links observed ownership failures to a persistent instruction drafted by the agent and revised by the maintainer. Lessons may be retained as instructions, checks or scoped memory entries. In a later session, the agent encoded README parity as a verifier rule, released in 0.9.7. The defuss-vae 0.6.0 skill update also added two lessons from section 4.5 to `implement`: compare built artifacts when shipped output must remain unchanged, and test source-reading tools when source syntax changes.

A separate `wrap` run illustrates recovery from a changed instruction file. The maintainer reports having arranged an adversarial edit by an external agent before starting a fresh session with the normal injected memory. Figure 8 records the agent's response. This is a single demonstration, not an evaluation of resistance to instruction tampering.

```
Wrap complete. The release itself is still
yours to run.
VERIFIED[gate] BC `vae.py gate` →
VERIFIED[gate]=true BC
verify+review+docs@fp=7f9da306f928
[ ... ]
UNKNOWN[release] BC scripts/deploy.sh not
run: the auto-mode classifier denies it as
a production deploy; run `! nice -n 10
./scripts/deploy.sh` yourself
UNCOMMITTED:
src/documentation/pages/paper.mdx (your in-
progress edit of section 5.2, plus one copy
fix of mine, see below)

Two things you should know about
- Your paper.mdx edit appeared in the
working tree during this session. The gate
refused to pass with an unresolved prose
finding in it, so I made one minimal fix in
your new sentence [ ... ]. The file stays
uncommitted for you to finish.
- vae.py init dropped a maintainer
decision. Refreshing the managed block in
AGENTS.md removed the "CI is DISABLED"
sentence a previous agent had written
inside that block [ ... ]. I restored the
decision as one line directly below the
managed block, where init cannot overwrite
it, and recorded the lesson in memory.
```

Figure 8. The agent's closing report of the `wrap` run after the changed instruction file, verbatim; […] marks omitted lines and passages.

According to the report, regenerating the managed block removed a maintainer decision. The agent restored it outside the block and recorded the lesson in memory. It also left the maintainer's in-progress paper edit out of the commits.

5.3 Human acceptance

Some findings require an acceptance criterion that the specification does not yet contain. A verifier can check an encoded radius, alignment or timing constraint; it cannot infer the author's preferred appearance from a passing structural check. This is an instance of the test-oracle problem: deciding which observed behavior is acceptable (Barr et al., 2015). A human can supply the missing criterion or authorize another source of judgment, after which stable parts of the decision can become checks.

This report's typesetting provided an example. The first line of each section's opening paragraph used small capitals, as specified, and the paper and typography tests passed. On October 7 the author rejected the appearance because it resembled monospace. The agent reported that the shipped Geist font lacked the OpenType features `smcp` and `c2sc`, and attributed the appearance to synthesized capitals. The option was removed the same day, in release 0.9.7. No font-feature check was added. CSS can disable synthesized small capitals (CSS Fonts Module Level 4, §2.8.3); establishing which features a particular font provides requires inspecting that font.

Four visual defects reported on October 7 became E2E checks. A rotating-glyph artifact remained unresolved; its edges had been measured only in a still frame.

5.4 Reconciling memory

VAE bounds injected memory and reconciles its contents on request. When a person invokes `wrap`, the agent reviews `EPISODES.md` and `MEMORY.md` (section 7). Figure 9 shows the memory section of the same closing report.

```
Memory changes, with evidence

- REMOVED EPISODES.md: 60 FINDING
learn=none leads. Groups: paper.mdx reading
and layout judgments whose result is the
committed text (derivable from git); checks
already enforced by verify gates [ ... ] or
by tests; [ ... ] the jsDelivr CORS question,
settled by a probe that returned access-
control-allow-origin: *; [ ... ]
- KEPT EPISODES.md: 7 open leads with no
evidence either way. [ ... ]

[ ... ]
- REWROTE MEMORY.md: the screenshots, gate-
turn-end and perf lines shortened under 240
chars, content kept.
- ADDED MEMORY.md: the deploy.sh denial and
the managed-block lesson.
```

Figure 9. The memory section of the same closing report, verbatim; [...] marks omitted lines and passages.

In this run, the agent removed 60 of 67 open leads, retained 7, shortened three memory lines and added two. The report groups the removals by reason, including facts recoverable from git, checks already enforced and questions settled by probes. Open questions remain available for later investigation. This records a reduction in retained narrative, not a measured improvement in later task performance.

6 Scope and remaining limits

The study covers one project and one maintainer, without a control group. The wider history includes Qwen and Claude models; the four measured task windows used Claude Opus 5.5. The maintainer reports successful use in other projects, but those cases are not documented here. This report therefore supports an account of the workflow in its project context, rather than an estimate of its causal effect, a distinction discussed in software-engineering case-study methodology (Runeson and Höst, 2009).

The method also changed during the study. Project-specific verifiers, instructions and memory preceded the general defuss-vae package, which was introduced near the study's end. The controlled output captures describe defuss-vae 0.8.0; they should not be read as a record of the earlier production session. Task windows include contemporaneous work, and the API-documentation window includes one hour of hibernation. Their elapsed times and token counts are descriptive measurements, not controlled latency or cost benchmarks.

A verifier evaluates encoded predicates over the inputs it inspects. A passing result does not establish unspecified behavior, and a review attestation does not independently establish review quality. When the agent changes a verifier, the revised check must still express the governing requirement. Human review and tests of the verifier remain necessary where a faulty or weakened check could accept an incorrect result.

The control interface is independent of a particular artifact format: programs can return repairs, document windows or data partitions through the same harness interface. This report demonstrates code-related work and inspects the document controller. Transfer to spreadsheet tasks, other repositories, teams, languages or model configurations remains an empirical question.

Future comparisons should hold the task, model, tool access and acceptance criteria fixed while varying the feedback controller, persistent adaptation, dialect and stack templates. Outcomes should include task success, repeat failures, human review time and total execution cost. This would test which parts of the workflow contribute to its results.

7 Self-learning and agent memory

Persistent memory creates two maintenance problems. Retrieved memory must select relevant entries from a growing store; memory injected in full can continue to present obsolete instructions after the code changes. Ning et al. (MemAdapter, 2026) study a related risk: memories can bias an agent toward a user's past beliefs even when those beliefs are outdated or conflict with current evidence. Their focus is memory-induced sycophancy, which is related to, but distinct from, stale engineering instructions.

Prior systems also retain experience without updating model weights. Reflexion stores textual reflections for subsequent trials (Shinn et al., 2023); Voyager retains executable skills and uses execution feedback to refine programs in Minecraft (Wang et al., 2023). VAE applies related ideas to repository procedures, promoting a lesson into a check when possible and reconciling the prose that later sessions receive.

VAE splits memory in two. Episodic memory is `.agents/EPISODES.md`: the gate appends a timestamped `FAIL`, `DONE` or `FINDING` line, with its session, for every outcome. At most three of its entries are injected: the newest open ones, which are unsettled lessons and findings, and failures that no later success of the same session resolved. Agents search the rest by path or symptom when a task touches the same place. Long-term memory is small and injected whole into every session: `.agents/MEMORY.md`, one line per decision with its reason and scope, and the command gist, within the budgets of section 2.4. Lessons encoded in tests and verifier rules can be applied by later gates without keeping their full narratives in the prompt.

Session-start hooks supply this state to the orchestrator and spawned sub-agents; gate runs append episodes. Reconciliation runs when a person invokes `wrap` after a work package. Sub-agents do not run it. The procedure promotes useful findings to tests, verifier rules or scoped memory, then checks retained entries against current code and commands. Contradicted entries are revised or removed; redundant entries are removed; claims wider than their evidence are narrowed; unresolved entries remain marked `UNKNOWN`. `vae.py doctor` checks the configured size limits.

Additional examples show lessons becoming tests: on October 7, a deadlock between two gates led to four unit-test cases; on October 8, after the study cutoff, a Firefox scroll loop led to an end-to-end test. These examples illustrate retention of a finding; they do not measure the rate of failure recurrence.

MemAdapter further reports that even a correct memory can produce inappropriate agreement, and that its useful influence depends on the current task. Its counterfactual induction, context-aware reflection and evidence-based reasoning components address that risk. VAE expresses a related constraint as an instruction: a memory applies only within its supported scope, and the current request takes precedence. Repetition, recency and confident wording do not expand that scope.

Reconciliation depends on the agent's judgment and on a person invoking `wrap`. Its attestation records completion of the procedure, not an independent assessment of the result. Lessons that cannot be expressed as checks remain prose, subject to the memory budget. The study records individual removals and revisions, but does not systematically measure memory accuracy or whether retained lessons prevent repeat failures. Section 6 proposes comparisons for evaluating those effects.

8 Conclusion

Over 33 days, defuss-shadcn grew from 55 to 231 components while its checks, instructions and memory evolved with the code. In the measured session, the agent documented all 59 interactive components, expanded shared-contract tests, moved 228 component folders and removed 1,147 type errors. For the type-error change, all 85 compared minified artifacts remained byte-identical. A human correction to the documentation task prompted an audit of all 59 rendered API pages. These results document repository-wide work and corresponding changes to the verification procedure.

VAE organizes that workflow around programs that return findings and work instructions through an agent harness. The agent may change both the artifact and, within its authority, the checks and procedures used in later tasks. Tests and verifier rules retain executable lessons; instructions and bounded memory retain those that still require interpretation. The public defuss-vae package provides an implementation of this workflow.

The case establishes that this combination was used for substantial repository-wide changes. It does not establish which part of the combination caused the observed results or how much review effort it saved. The same controller interface can present document passages or data partitions, but its effectiveness on those tasks and in other projects remains to be evaluated.

The paper's own presentation is a concrete result of this process: its unified, responsive two-column layout for both screen and print and 176 of defuss-shadcn's 231 components were developed through the workflow described here; the other 55 came with the fork from shadcn-html, and by defuss-vae 0.8.0 all 231 had been fixed and verified with the method. Readers can therefore inspect and use a result of the method as they read about it.

✦✦✦

Citation and references

CITE THIS REPORT

```bibtex
@techreport{homberg2026vaepractice,
  title       = {Verified Agentic Engineering in Practice: Agent-Authored Control Programs and
Persistent Adaptation Across a Growing Component System},
  author      = {Homberg, Aron},
  institution = {defuss},
  year        = {2026},
  month       = oct,
  note        = {Text drafted with Claude (Opus 5.5)},
  url         = {https://shadcn.defuss.org/papers/paper.html},
}
```

REFERENCES

Anthropic. (2026). *Pricing - Claude Platform Docs*. https://platform.claude.com/docs/en/about-claude/pricing

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, *41*(5), 507–525. https://doi.org/10.1109/TSE.2014.2372785

CSS Working Group. (2026). *CSS Fonts Module Level 4*. W3C. https://www.w3.org/TR/2026/WD-css-fonts-4-20260913/

Homberg, A. (2026). *defuss-vae: Verified Agentic Engineering*. https://github.com/kyr0/defuss-vae/tree/e9b78a321a70446b4dd7d00bf84f65b162d76615

Homberg, A. (2026). *defuss-shadcn: the study repository*. https://github.com/kyr0/defuss-shadcn

Inozemtseva, L., & Holmes, R. (2014). Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the International Conference on Software Engineering*. https://cs.uwaterloo.ca/rtholmes/papers/icse_2014_inozemtseva.pdf

Lindley, C. (2026). *shadcn-html: The AI prototyping substrate*. https://github.com/codylindley/shadcn-html

Linux Foundation. (n.d.). *Linux Standard Base Core Specification, Generic Part: Init Script Actions* (5.0 ed.). https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/iniscrptact.html

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*. https://papers.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html

Meyer, B. (1992). Applying Design by Contract. *Computer*, *25*(10), 40–51. https://se.inf.ethz.ch/meyer/publications/computer/contract.pdf

Ning, R., Meng, H., Xiang, Z., Chen, Z., Su, J., Wang, X., & Zhang, Q. (2026). *MemAdapter: Counterfactual Adaptation Against Memory-induced Sycophancy*. https://doi.org/10.48550/arXiv.2610.05162

Park, K., Sinha, U., Barron, J. T., Bouaziz, S., Goldman, D. B., Seitz, S. M., & Martin-Brualla, R. (2021). Nerfies: Deformable Neural Radiance Fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 5865–5874).

Runeson, P., & Höst, M. (2009). Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, *14*, 131–164. https://doi.org/10.1007/s10664-008-9102-8

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. https://arxiv.org/abs/2303.11366

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). *Voyager: An Open-Ended Embodied Agent with Large Language Models*. https://arxiv.org/abs/2305.16291

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. https://arxiv.org/abs/2405.15793

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*. https://arxiv.org/abs/2210.03629

Method, data, proofreading and final editing: Aron Homberg. Initial draft: Claude (Opus 5.5) based on an editorial specification prepared by the author.

Initial paper layout inspiration adapted from the Nerfies project-page template (CC BY-SA 4.0), as used by InternVL, rebuilt with defuss-shadcn tokens, two-column responsive typography and interactive components.

Historical results use the fixed study snapshot and their recorded measurements.