WHITEPAPER / VERSION 2.0

# Semantic Surface Architecture

A naming system for codebases built with AI: structure your code so agents understand it and you stay in control

Michael Latulippe

Version 2.0 / Updated August 6, 2026 / CC BY 4.0

www.michaellatulippe.com/whitepaper/semantic-surface-architecture

# In this paper



If you only read one section, read Section 6. It is the how-to. Everything before it is why it works. The whole paper is about 35 minutes. Section 6 is six of them.

> *The whole method in five lines. Pick a theme for your app: a ship, an inn, an arcade. Name three to five systems from it, one common word each: Vault for money, Gate for login. Write one sentence per name in a file called CLAUDE.md in your project root. Keep the folder names matching the system names. From then on, say "the bug is in Vault" instead of re-explaining your app every session.*

# 1. The Problem

Your AI coding assistant forgets your architecture every session.

You close the laptop with a model that finally understands how your payments flow works. You open it the next morning and that understanding is gone. The context window, which is the fixed amount of text a model can hold in working memory at one time, has been cleared. So you start over. You paste the file tree, explain which service owns what, and repeat the rule the two of you worked out yesterday. Fifteen minutes, sometimes thirty, in my own sessions, before any real work begins. Call it the re-explaining tax. Most people building software with AI pay it several times a day and have stopped noticing.

For someone building their first real system this way, the tax compounds into something worse. That was my own six-week arc, and I have watched enough people describe the same one that I no longer think it is mine alone.

Week one is euphoric. You describe a feature in plain English and it appears. Week three you are shipping faster than people who have done this for a decade. Then it turns. Six weeks in, you have 40,000 lines of code you did not write. You know what the app does because you use it. You cannot say what anything is called. There is a file that handles payments and you are fairly sure there is a second one, and you do not know which is live. Every fix breaks two other things. You have started avoiding whole directories. When the AI asks which service should own a new function, you have no answer, so you let it decide, and the pile grows another layer.

That is not a personal failure and it is not a sign you should have studied computer science first. It is a missing vocabulary. You own a system nobody, including you, can name.

The obvious fix was supposed to be more room. Frontier models from the Claude, GPT, and Gemini families now offer context windows measured in the hundreds of thousands to millions of tokens, orders of magnitude beyond what was normal three years ago. Benchmarks like SWE-bench Verified, which measure whether a model can fix a real GitHub issue, have seen top scores climb so far that headroom, not capability, is the discussion. The bottleneck moved. It now sits in long-horizon navigation and system understanding, which is the unglamorous work of figuring out what matters in a large repository.

One 2026 study of agents with huge context windows found they did not fail because they ran out of room [1]. They failed because they never discovered that a relevant file existed. The question stopped being “can the model see this file” and became “does the model have any reason to look.”

Chroma's 2025 work on context rot points the same direction from the other side: models do not use their context uniformly, and their performance grows increasingly unreliable as input length grows, even on simple retrieval tasks [2]. A bigger window is not a bigger brain. It is a bigger room with the same amount of light.

Salience is a naming problem. When an agent scans a file, the identifiers are the densest signal in it. Function names, class names, directory names, and error names are short, repeated, and human-chosen, so they carry more intent per token than anything else on the page. A file called `utils/helpers.ts` full of functions called `processData` tells an agent nothing worth acting on. A file called `lib/systems/vault/charge.ts` full of functions called `Vault.Charge.create` tells it what the file is for, what it is allowed to touch, and where the neighboring pieces live.

Semantic Surface Architecture is a system for making those names do that work on purpose. You organize your code by who is asking, which SSA calls Surfaces, and by what capability is being invoked, which SSA calls Systems. The system names come from a theme, a small metaphor world, so they hang together and stay memorable. One grammar runs everywhere: `System.Component.action()` for calls, `System.Component.ErrorName` for failures, `System.Event` for things that happened. You keep the directory tree matching the vocabulary, so a path is a claim about meaning. And you enforce a handful of separation rules that a single grep over your import statements can verify. That last part is the difference between a style guide and an architecture.

The payoff is small and constant. You say "the bug is in Vault" and the agent goes to the right four files. An error in your logs that reads `Vault.Refund.InsufficientFunds` tells you which system, which piece of it, and what went wrong, before you open anything. A pull request written by a model at 2am can be judged in ten seconds on whether it belongs where it was put.

This paper is written for two readers. You might be six weeks in with no formal training. Everything here is written so you can follow it, and jargon is defined the first time it appears. Or you might be the engineer who has heard "use good names" a thousand times and wants to know what is actually being claimed and on what evidence. Section 2 is for you, including the parts of the evidence that cut against a simple story.

SSA was first published in March 2026. This is version 2.0, written in August 2026, revised for what the five months since taught me: the research got sharper, other people arrived at similar conclusions independently, my own two codebases changed in ways the first version did not predict, and the file tree quietly became a routing mechanism for what an agent knows.

# 2. What the Evidence Says

A naming system is a strong claim dressed as a soft one. It says the words you choose change what a machine does with your code. That is testable, and over the past two years people have tested pieces of it. This section walks through what they found, including the parts that complicate my case. None of these studies evaluated SSA. They establish the premises SSA is built on, which is a smaller and more honest thing to say.

## Big windows did not fix it

Paipuru measured this directly [1]. On hidden-dependency tasks, where the file you need is not lexically related to the file you start in, an agent given graph-structured dependency navigation reached 99.4% architectural coverage. The same agent without it reached 76.2%. Keyword retrieval was near-perfect on tasks where the right words appear in the right file, and close to useless where they do not: it moved hidden-dependency coverage from 76.2% to 78.2%. One more detail worth keeping: in 58% of trials where the graph tool was available, the agent never called it. A structure the agent must opt into is a structure it often ignores. Paths and names are structure the agent cannot avoid reading.

The obvious reading of that study is that the fix is a tool, not a vocabulary. Install a dependency graph and skip the renaming. That reading is fair, and the two are not rivals: a graph tells an agent what connects to what, and a name tells it which of those things is worth opening.

## Names carry a large share of the signal

The first serious probe was Wang and colleagues, first posted in 2023 and revised through 2024 [3]. They took code, systematically stripped identifier names, and measured what models could still do. In Java code search, mean reciprocal rank, a standard measure of whether the right result comes back near the top, fell from 70.4% to 17.4% once every class of identifier was anonymized. Anonymizing variable names alone barely moved it, from 70.4% to 67.7%. The collapse came from the method and function names. The models were still looking at the same logic. They had lost the labels that mattered.

Two caveats matter, and I would rather state them than have a reader find them. That work used encoder models, older systems that read code rather than write it (CodeBERT and GraphCodeBERT), on search and duplicate-finding tasks. It did not test an agent editing a live repository, which is what most of us are doing now. So read it as evidence about code understanding in general, not as a direct measurement of your Tuesday afternoon.

Le and colleagues brought it closer in 2025 [4]. They applied semantics-preserving obfuscation, meaning they changed names without changing what the code does, and then measured GPT-4o on class-level summarization using the ClassEval benchmark. Accuracy fell from 87.3% to 58.7%. More interesting to me: performance also degraded on execution prediction, a task that in principle depends only on structure. If a model can trace what code does, renaming a variable should not matter. It mattered.

The authors read their result partly as a critique of benchmarks, arguing that scores reward memorized naming patterns as well as real comprehension. That reading is fair and I will not soften it. The practical corollary stands either way: identifier names carry a large share of the intent signal models actually use. The effect is task-dependent. Some tasks barely move. Do not let anyone, including me, tell you that stripping names universally breaks models.

## Wrong names are worse than no names

The 2024 and 2025 work asked what happens when names go missing. In 2026 the question got meaner: what happens when names are present, fluent, and wrong.

Le, Nguyen, and Nguyen tested adversarial renaming, where identifiers are meaningful but misleading [5]. A function that deletes records is called `archiveRecord`. A cache is called a queue. When misleading names were combined with real semantic disruption, models stopped hedging and produced high-confidence wrong answers. The same paper reports something that should change how you review AI output. Coder-tuned and instruct-tuned models, the kind most people run for code, showed no significant correlation with human difficulty on the same passages. Reasoning-tuned models aligned moderately. So the intuition "this name would not confuse me either" is a usable test only against a model that reasons, and not against the coder-tuned models doing most of the world's autocomplete.

Guzmán Lorenzo's case study on poisoned identifiers shows the mechanism up close [6]. Misleading identifiers propagated into reconstructed code in every baseline run. In 15 of 17 runs, the model wrote the wrong name while correctly describing the operation in its own comments. It knew what the code did. It repeated the lie anyway, because the name is what gets carried forward. Explicit verification prompts, telling the model to check names against behavior, failed in all 12 attempts. Reframing the task helped where instruction did not. This is a small study, one model family, and I cite it as illustration rather than proof. But the shape of the failure is worth sitting with: you cannot prompt your way out of a bad name, because the prompt and the name are competing for the same job and the name is inside the artifact.

The strongest result of the three is peer-reviewed. Haroon and colleagues, at ICST 2026, applied semantics-preserving mutations, including identifier renaming, to code where models had already correctly localized a fault [7]. After mutation, the models failed to find the same bug in 78% of cases, across 750,000 localization

tasks and ten models. The mutation set is broader than naming. It includes dead code and misleading comments, and misleading variable names are one of the six. The paper does not break the 78% out by mutation type. What it establishes is that fault localization rides on surface form rather than semantics. Same bug. Same logic. Different surface.

Put the three together and you get one sentence worth keeping. In an AI-assisted codebase, a wrong name is not cosmetic debt. It is an active instruction, followed confidently, that you cannot retract with a comment.

## Why documentation does not rescue this

The standard response to all of the above is to write it down. Add a `CLAUDE.md`, the file Claude Code reads at the start of every session, or an `AGENTS.md`, the same idea read by other agents. Explain the architecture in a file the agent reads at session start. I do this. I recommend it in Section 6. It is also not the load-bearing wall that people treat it as.

Khatri ran the test nobody had run [8]. Two coding agents, 17 real tasks across 3 real repositories, 288 evaluated runs. The only thing that changed was the context-file strategy. Correctness did not measurably move on either agent. The study is small, and its author is careful to call the bound descriptive: it makes a large improvement look unlikely, it does not prove zero. But the failure analysis is the part to sit with. When runs failed, the agent had picked a bad approach or wired the pieces together wrong. It was not missing facts about the repository that a context file would have supplied.

I read that as support for what SSA argues, and I want to be explicit about why, because it can be read the other way. If descriptive files about your architecture do not move correctness much, one conclusion is that architecture description is useless. The better conclusion is that annotation cannot rescue an illegible substrate. A context file describes names and boundaries. The names and boundaries themselves live in the code, and that is where the agent is working. Writing a beautiful paragraph about how `DataService` is really the payments system does not change the fact that every file the model touches says `DataService`.

There is a maintenance argument too. Chatlatanagulchai and colleagues studied 2,303 context files across 1,925 repositories and found they grow by frequent incremental additions and evolve like configuration rather than like documentation [9]. That is a fair description of every one I have owned. Vasilopoulos documents the far end of the curve in a single-developer case study where context infrastructure grew to roughly a quarter of the codebase [10]. I cite that as an existence proof of the burden, not as a typical figure. Anthropic's engineering guidance lands in the same place from the vendor side: treat context as a finite resource and curate it [11]. Every token you spend explaining your architecture is a token not spent on the problem, and it must be re-spent every session, forever.

Names do not have to be re-spent. They are already in the file.

## Compression is a false economy

An objection I hear from people who watch their API bills: longer names cost more tokens.

Ustynov tested compression directly, on log formatting [12]. Aggressively compressing the same information, abbreviated codes plus a schema header, cut input tokens by 17%. Total session cost rose 67%, because the model burned reasoning tokens decoding the terse forms before it could work. That is a result about log events, not identifiers, so read it as a mechanism rather than a direct measurement of naming. The mechanism is the one that matters: tokens saved on the page get paid back with interest in reasoning. The same paper argues the naming point directly: token savings should come from cutting zero-information structural tokens, not high-information semantic ones.

SSA's names are short at the System layer and long in use. `Vault` is one word. `Vault.Refund.InsufficientFunds` is not. The unit that reaches the model is the full form, and it is specific enough to read once instead of read once and puzzled over. Ustynov goes further than I do. He argues machine representation can decouple from human readability. I want one vocabulary both readers share.

The last piece is a warning against a trap I fell into personally. Jin measured what happens when you give an agent a formal description of your architecture, and found it cut exploration steps by 33 to 44% [13]. Then came the useful part: the format made no difference. S-expressions, a code-like text format, plus JSON, YAML, and Markdown all performed the same. The information matters. The notation does not. If you find yourself designing a schema for your architecture file, stop and go name a system instead.

## What this section does not prove

None of these studies tested Semantic Surface Architecture. Nobody has run a controlled trial where one team builds with SSA and another builds without it. That trial would be expensive and I have not run it.

What the literature supports is narrower and still worth having. Identifier names carry a large share of the signal models use. Misleading names cause confident errors that prompting does not repair. Model confusion does not track human confusion, so your own comprehension is a poor proxy. External context files do not compensate for an illegible codebase, and they cost real maintenance. Descriptive identifiers are economically cheaper than compressed ones. Structural information helps agents navigate, and its format is irrelevant.

Those six findings are the ground SSA stands on. What SSA adds is a specific system for acting on them.

# 3. The Framework

SSA has five layers. Every name in your codebase belongs to exactly one of them, and each layer has its own naming style. The whole framework fits in a table, and I would rather you memorize the table than the paper.

| Layer | Answers | Named like | Example |
|---|---|---|---|
| Surfaces | Who is asking | Single evocative word, one per actor | Market, Helm, Wire, Cortex |
| Systems | What capability is invoked | Single evocative word, drawn from a theme | Vault, Lease, Signal, Beacon |
| Components | How a system is organized inside | Noun, always prefixed by its System | Vault.Refund, Lease.Reserve |
| Actions | What operation is performed | Verb from a closed set, lowercase | `Lease.Reserve.create()` |
| Outcomes | What happened | Past-tense events, descriptive errors | `Vault.Refund.InsufficientFunds` |

A Surface is defined by who is on the other side of the screen or the socket. Not by technology. A public marketplace browsed by strangers, a dashboard used by signed-in owners, an API consumed by machines, and an admin console used by you are four different Surfaces even when they share a database, a framework, and a deploy pipeline. Organizing by actor is the part that senior engineers push back on first, and it is the part that pays fastest with an agent, because "who is this for" resolves permission questions, copy questions, and layout questions in one move.

A System is a capability, named as a thing rather than as a job title. Payments are Vault. Rental lifecycle is Lease. Real-time events are Signal. Systems are the nouns you will say out loud fifty times a week, so they need to be short, distinct, and pleasant to say.

Components, Actions, and Outcomes are where the grammar does its work, and they mostly write themselves once the first two layers exist.

Mostly, but not entirely, so here is the test for a Component. Ask what distinct jobs the System does that you would explain separately to a person. Hearth's Purse holds balances, settles debts, and records payments: Purse.Balance, Purse.Settle, Purse.Record. If you cannot name a second job, the System has one Component and that is fine. Components are discovered, not designed. Add them when the code forces the distinction.

## Start with a theme, not with a list

The single most useful move in adopting SSA is also the least technical. Before you name anything, pick a metaphor world.

If your app were a place or a machine, what would it be? A ship. An observatory. A workshop. A city. A kitchen. An arcade. A garden. It does not matter which, and there is no correct answer for a given domain. What matters is that one world generates many consistent names, so you are never naming things one at a time from scratch.

The clearest example I have is the owner dashboard on one of my platforms, a Surface called Helm. Helm is the ship's wheel, and once that word was chosen every view under it named itself. The overview is Bridge. Active rentals are Voyages. Owned domains are the Fleet. A single domain is a Vessel. The listing wizard is the Dock, because that is where a vessel gets prepared. Earnings are the Treasury. Agent management is the Crew. Settings are Quarters.

Nobody sat and brainstormed eight names. The theme produced them, and it will produce the ninth when I need it, which is the point. A theme is a name generator with a consistency guarantee. It also makes the vocabulary memorable, and memorable matters more than it sounds: you will only stay consistent with names you enjoy using.

A theme holds best inside one Surface. Helm produced Bridge, Voyages, Fleet, Vessel, Dock, Treasury, Crew, and Quarters without a single brainstorm, and it will produce the ninth. Across a whole platform it holds less well than I expected. Popdot's systems draw on several metaphor worlds: nautical, electrical, heraldic, optical. The clusters are coherent and the whole is not, and that is what a real vocabulary looks like after a year of shipping. The guarantee a theme gives you is local. Do not promise yourself one world for the whole system. I did, and I did not get it.

Two warnings. A theme should be concrete enough to have parts. "Ocean" is weak. "Ship" has a wheel, a hold, a crew, a log, and a dock. And a theme should not be a joke you will resent in a year. You are going to type these words several hundred times.

## A worked example

Here is a small app, invented for this paper. Hearth is a tool for people sharing a house who need to track who paid for what and settle up at the end of the month. Roughly 6,000 lines. Built in three weeks by one person with an AI agent. This is what it looked like before.

```
src/
  services/
    PaymentService.ts      processData(), handleRequest(), doUpdate()
    UserService.ts         getData(), processData()
    DataService.ts         handleRequest()
  utils/
    helpers.ts
    helpers2.ts
```

Nothing here is unusual and nothing here is wrong in the way a broken test is wrong. It is just mute. `PaymentService.processData()` tells an agent that something happens to something. `helpers2.ts` tells it nothing at all. Ask a model to fix a bug in settlement and it has to read most of the repository to find out where settlement lives, which is exactly the salience failure from Section 1.

Now the same app with a theme. The house is an inn.

Surfaces, meaning who is asking:

**Hall.** The shared space every housemate uses.

**Keep.** The one person who administers the house account.

**Wire.** The mobile client, which is a machine talking to the server.

Systems, meaning what capabilities exist:

**Slate.** The running record of who owes what. The thing you chalk a tab on.

**Purse.** Money actually moving between people.

**Roster.** Who lives here, who moved out, who is a guest.

**Bell.** Notifications.

**Larder.** Receipts and uploaded files, stored where they keep.

Five systems for a small app. That is the right number. The grammar then falls out:

```
Slate.Charge.create()              a housemate records a shared expense
Slate.Charge.Created               the event other systems can listen for
Purse.Settle.process()             month-end settlement runs
Purse.Settle.InsufficientFunds     the failure you will actually see in logs
Roster.Member.verify()             confirm a new housemate's email
Hall.Slate.Split                   the screen where you split a charge
```

And the file tree stops being a filing cabinet and starts being a map:

```
app/
  hall/slate/[id]/split/       ->  Hall.Slate.Split
  keep/members/                ->  Keep.Roster
  api/wire/slate/              ->  Wire, Slate endpoints
lib/
  systems/
    slate/charge.ts            ->  Slate.Charge
    purse/settle.ts            ->  Purse.Settle
    roster/member.ts           ->  Roster.Member
```

Same app. Same 6,000 lines. Now "the settle job is double-charging people who moved out mid-month" points at `lib/systems/purse/settle.ts` and `lib/systems/roster/member.ts`, and both you and the agent know it without opening anything. That sentence is also something a person with six weeks of experience can say with confidence, which is the part I care about most.

## The naming rules

Seven rules, in order of how much they matter.

**One word per System, strongly preferred.** Vault, Slate, Purse, Bell. Single words stay memorable, fit in paths, and never invite an abbreviation. This is a strong preference rather than a law, and Section 4 shows the honest counterexample from my own code.

**Evocative over descriptive.** `Vault` beats `PaymentProcessingService`. A vault already means secured value to any reader, human or model, without a line of documentation. Descriptive names describe the implementation, which changes. Evocative names describe the intent, which does not.

**Never reuse a word for two things.** If Signal is real-time events, nothing else in the codebase is allowed to be a signal. Ambiguity is the one failure mode with no cheap fix.

**Check the name against the ecosystem you build in.** Before a word enters the vocabulary, search it against your package manager and your framework. A model's prior for `Relay` is a GraphQL client and its prior for `Signal` is a reactivity primitive. Borrowing a famous name imports its meaning, and Section 2 is about what a wrong meaning costs. I learned this late. Popdot's Relay routes subdomains and its Signal carries events, and both names fight priors they should not have to fight.

**Components are nouns and always carry their System.** `Vault.Refund`, never a bare `Refund`. The prefix is what makes a name portable into a log line, an error, a commit message, or a sentence you say out loud.

**Actions come from a closed verb set.** create, read, update, delete, verify, process, cancel, submit, approve, reject. A closed set means an agent can predict the method name before reading the file, and predictable names are findable names. When something genuinely does not fit, add a verb to the set deliberately and write it down. Do not invent one per file.

**Outcomes are past-tense events and specific errors.** `Lease.Reserve.Completed`. `Vault.Refund.InsufficientFunds`. An error name should be readable by a person who has never seen your code, because at 3am that person is you.

## The rules a grep can check

Naming discipline decays unless something checks it. SSA's separation rules exist to be mechanical:

- Surfaces never import each other.
- Systems never import Surfaces.
- Foundation systems import nothing above them.
- Dependencies point only downward: orchestration imports domain, domain imports foundation.
- Internal names never reach user-facing copy.

Systems sit at one of three tiers. Foundation systems depend on nothing inside your code: logging, storage, auth checks. Domain systems own a capability and may use foundation. Orchestration systems coordinate other systems and may use both. On SAP, Scribe and Shield are foundation, Vault and Ledger are domain, Crank is orchestration.

Those five lines are the whole architecture, and they are verifiable with a single grep over your import statements. Grep is the command-line tool that searches files for a pattern; the point is that no new tooling, no build step, and no service is required.

```
grep -rn "from '@/surfaces/" lib/systems/ && echo "RULE BROKEN: a system imports a surface"
```

If that prints a file, the rule is broken, and the file name tells you where. Dependency-cruiser, ArchUnit, and import-linter do this properly, with cycle detection and transitive checks a grep cannot reach. Use one if you have it. The grep's advantage is that it needs no adoption decision. It costs one line and it runs today.

If a file under your Market surface imports something from your Tower surface, the grep prints it and you have a bug in the structure before you have a bug in production. One of my two platforms runs this check as part of merging any change, so a violation gets caught by a machine at review time rather than by me in six months.

This is where SSA stops being advice. Everyone in this space says to use good names. A rule you can grep is a different kind of object: it either holds or it does not, and the answer takes one second.

## Structure now decides what an agent knows

The last piece is new since March, and it changed how I think about directories.

Agent tools now load instructions from the file tree itself. Claude Code reads skills from nested `.claude/skills/` directories, so when the agent touches a file in a subtree, the skills in that subtree activate [14]. The Agent Skills open standard that defines this format is published at agentskills.io, whose client showcase lists more than forty products, including Claude Code, Codex, Cursor, Copilot, and Gemini CLI, and describes that list as partial [15]. Your directory layout is now a routing mechanism for what the agent knows, and when.

That has a practical consequence. If your systems live in coherent directories, you can attach guidance to a system and have it appear exactly when an agent works in that system, and stay out of the context window the rest of the time. If your code is organized as `services/` and `utils/`, there is nothing to attach guidance to, because the directory does not correspond to a concept.

The same shift is visible from another direction. At least one agent platform, Hermes, now writes its own procedural documentation from workflows that succeeded [16]. When that happens, your codebase names propagate into machine-written docs that other agents will read later. Whatever you called it is what it will be called from now on, in places you did not write and will not review.

If you came here to fix your own project, you now have enough. Jump to Section 6 and start. Sections 4 and 5 are the evidence trail: what happened when this ran for a year on two real systems, and where it sits against everything else.

# 4. Two Production Systems

What follows is a report from the builder of two live systems, not a controlled study. I designed the framework, I applied it, and I am the one telling you it worked. Read Section 7 before you believe any of it too hard. What this section can honestly offer is detail: the actual vocabularies, the actual changes over five months, and the places where my own rules bent.

Both platforms were built primarily through AI-assisted development by one developer, using SSA throughout. Together they run 7 surfaces and 40 named systems across two domains that have almost nothing in common: autonomous agent commerce and vacation rentals.

## Popdot AI: when your customers are machines

Popdot AI is a subdomain rental marketplace built for autonomous AI agents. An agent discovers the platform machine to machine, is verified, rents a live HTTPS address for somewhere between an hour and a month, pays per transaction in USDC, a digital dollar, and deploys content to it. On the other side, humans list domains they own and earn most of each rental. It has been live since July 2026, and the agent surface came first.

That last detail changes the stakes of naming. On a normal product, your vocabulary is read by you, your teammates, and whatever coding agent you work with. On Popdot, the names in the codebase also appear in agent-facing manifests and documentation, so they are read by machines at runtime, by customers, in production. A confusing system name is no longer an internal ergonomics problem. It is a product defect that shows up in someone else's agent loop.

Popdot has 4 Surfaces and 22 Systems.

The Surfaces are Market, the public marketplace for humans; Helm, the authenticated dashboard for people who own domains, named for the ship's wheel; Wire, the API where agents live, named for the electrical connection; and Cortex, the administrative surface, named for the neural center. Helm's nautical views are the theme example from Section 3: Bridge, Voyages, Fleet, Vessel, Dock, Treasury, Crew, Quarters.

Twenty-one of the twenty-two systems are named. Eight of them, the ones the transfer analysis later turns on:

| System | What it does |
|---|---|
| Gate | Human authentication |
| Vault | Money movement |

| System | What it does |
|---|---|
| Sigil | Agent identity |
| Prism | Pricing |
| Orbit | Search |
| Signal | Real-time events |
| Beacon | Analytics |
| Echo | Agent discovery and marketing |

The full list of 22, including Lease for the rental lifecycle, is in the appendix. You can read it and know roughly what the product does, which is the test.

There is a pile of admin plumbing under Cortex that never earned a name. It should have one. That it does not is the framework's own rule going unenforced in my own repository.

The grammar in live use looks like this:

```
Helm.Voyages.Deploy            Surface.View.Panel
Lease.Reserve.execute()        System.Component.action()
Vault.Refund.InsufficientFunds System.Component.Error
echo.trace                     System.Event
```

`execute` is not in the starting verb set. I added it when reservation needed a verb that was not `create`. Additions to the set are allowed. They have to be deliberate, and they have to be rare.

And the file system carries the same information. In Hearth, `app/hall/slate/[id]/split` is `Hall.Slate.Split` and Purse lives in `lib/systems/purse/`. Both my platforms follow the same shape: the surface segment leads, and the system directory matches the system name. Nothing needs to be explained, because the path already said it.

On Popdot, the renter-facing path says rentals and the vocabulary says Voyages. URLs are user-facing and the vocabulary is internal, so they can pull apart. Renters read "rentals." I read "Voyages." Where they diverge, the rule is that the URL segment carries the user's word and the mapping lives in one place. Where you can keep them identical, do, because those are the parts that grep cleanly.

## Sleep Around Points: the same framework, moving real money

Sleep Around Points is a Disney Vacation Club points rental marketplace. Owners who have unused points list them, renters who want a discounted Disney resort stay book them, participants are identity verified, payments are managed by the platform, and the booking process is handled through the platform. It is live in open beta with real users and real money.

It has 3 Surfaces and 19 Systems.

The Surfaces are Market, the marketplace where all users live, with owner features appearing conditionally inside one unified dashboard; Tower, the administrative surface, named for air-traffic control watching the field; and Wire, the API.

Eight of the nineteen Systems, matched against the Popdot table above:

| System | What it does |
|---|---|
| Gate | Auth and identity verification |
| Vault | Money movement |
| Deed | DVC contracts, a real-estate metaphor |
| Prism | Pricing and point charts |
| Orbit | The trip request board, where renters post what they want |
| Signal | Messaging |
| Beacon | Conversion events |
| Compass | First-party analytics |

The full list of 19 is in the appendix.

SAP is also where the separation rules from Section 3 became procedure rather than intention. An SSA conformance review runs as part of the merge process: before a change lands, it gets checked for whether the diff respects the naming and separation rules.

## What transferred, and what did not

Applying the same framework twice, in domains this far apart, is the closest thing to an experiment I have. Here is what actually happened to the vocabulary.

**Three names transferred with the same meaning.** Gate, Vault, and Prism mean the same thing on both platforms: identity, money, price. Those three sit at the boundary between the product and the outside world.

**Three names transferred and changed scope.** Orbit is search on Popdot and a trip request board on SAP. Signal is real-time transport on Popdot and messaging on SAP. Beacon is analytics on Popdot and conversion events on SAP, with analytics moving to a new name, Compass. All three drifted toward whatever the second product needed. That is a weaker result than a clean transfer, and it is the one I got. A name at the right level of abstraction survives the move. It does not always arrive with its scope intact.

**Several capabilities transferred under different names.** Popdot's Pulse and SAP's Crank both run scheduled jobs. Popdot's Bloom and SAP's Ping both send email. Popdot's Lease and SAP's Ledger both own the transaction lifecycle. Different themes made different words natural, and forcing a shared dictionary would have made both codebases slightly worse. What transfers is the pattern, not the words.

**The same is true one layer up.** Popdot's admin surface is Cortex, the neural center. SAP's is Tower, air-traffic control watching the field. Identical role, different metaphor, because the products live in different worlds. The layer transfers. The name is chosen per product. If you take one thing from this section, take that.

**Some systems could never transfer.** Sigil, Echo, and two of the security systems exist only because the customers are AI agents. Deed, Atlas, and Pact exist only because the product is real-world lodging with contracts and resorts. That is roughly one system in six on each platform, and it is the healthy part, not the leftover part.

## Three things that went differently than the first version predicted

The March paper described SAP with four surfaces, including a Helm. **Helm is gone.** It was merged into Market, because owners and renters turned out to be the same people wearing different hats, and owner features now appear conditionally inside one dashboard. The surface count went down. I want to be plain about this, since a framework author's instinct is to hide it: the user reality did not justify the split, so the split was removed. SSA vocabularies shrink as well as grow, and a shrinking vocabulary is not a failure of the framework.

**A name was retired cleanly.** A Popdot system named Gate (Content) was taken out of the tree. Because the system had a name and a directory, retiring it was a bounded operation: the name left the vocabulary, the code left the tree, and nothing else moved. Compare that to deleting a concept that was spread across six files called `helpers`.

**My rules bend in four places, and I would rather list them than have you find them.** One security system's name is two words and stays, because a good two-word name beats a bad one-word name. CAPI is worse: an acronym, from no theme, and it should be renamed. Trial is the literal descriptive word in a vocabulary that argues for evocative ones. And for a while Popdot ran two systems called Gate at once, authentication and content gating, disambiguated by a parenthetical, which is exactly the ambiguity the rules forbid. Retiring the content one was clean partly because it resolved a collision I should not have created. A rule set with no violations in production is a rule set nobody has used.

## Echo, and naming for a machine audience

Echo is the newest system, added in July 2026, and it is the best illustration I have of a name doing architectural work.

The founding metaphor is echolocation. Agents perceive the internet roughly the way a bat perceives a cave. They emit probes and build their picture of the world from what comes back. To a machine, the reflection of the service is the service. There is no landing page, no brand recall, no word of mouth. There is only what returns when something asks.

Once that framing existed, the components came quickly: Pack for agent-facing metadata, Card for the machine manifests, and four more that all answer the same question.

The metaphor did more than generate six words. It told me what Echo was responsible for and, more usefully, what it was not. Echo owns everything about how the platform appears to a machine that is asking. It does not own the transaction, which is Lease, or the money, which is Vault. When you can state a boundary in one sentence and the name agrees with the sentence, the agent working in that directory tends to stay inside it.

## The Cortex Parity Principle

Autonomous agents behave in ways their operators do not anticipate, which is consistent with findings from independent red-team research on autonomous agents [17]. A platform whose paying customers are agents has to assume that.

Popdot's answer is a governance rule stated in the vocabulary itself, the Cortex Parity Principle. Anything an agent can do through Wire must be visible to administrators, controllable through admin intervention, and auditable. In practice it means a new agent capability is not finished when the Wire endpoint ships. It is finished when the Cortex view ships with it.

I include it here because it shows what SSA is for beyond ergonomics. The principle is one sentence long only because Wire and Cortex are named things. Without Surfaces, the same rule would be a paragraph of hedged prose about API endpoints and admin panels, and nobody would be able to tell whether a given pull request satisfied it.

## The scale I am reporting from

Both platforms are young and neither is large. Popdot went live in July 2026. Sleep Around Points is in open beta with real users and real money. I am not reporting from scale. I am reporting from two codebases one person can still hold in his head, which is the condition SSA was designed for, and honestly the only condition it has been tested in.

# 5. Related Work

SSA did not arrive in an empty room. This section places it next to the things you have probably already read.

## AGENTS.md

If you use coding agents at all, you have met `AGENTS.md`, the convention of putting a markdown file at the root of a repository telling agents how to work in it. The site reports over 60,000 open-source projects using it, a figure it derives from a live code search, so treat it as an order of magnitude rather than a count. Since the Linux Foundation announced the Agentic AI Foundation on December 9, 2025, AGENTS.md has been one of its anchor projects [18].

The important design decision in `AGENTS.md` is that it is deliberately schema-free. There is no required structure, no field list, no validator. That is why it spread, and it is also why every one you read is different. Some are build instructions. Others are style guides, or 400 lines of architecture prose that went stale in March.

SSA is complementary to it, and I would put the relationship this way: SSA is what makes your `AGENTS.md` short. If your systems are named and your separation rules are greppable, the file has almost nothing left to say. A vocabulary table, a few rules, a pointer to where things live. If your codebase is illegible, the file has to carry the entire architecture in prose, which is the case where Khatri's study found no measurable gain [8]. The container is fine. SSA supplies the discipline the container leaves open.

## Existing naming guidance

Here is the claim I am prepared to defend, and I want it narrow. The naming guidance I have been able to find in this space, including AGENTS.md, Adam Tornhill's Code for Humans and Machines newsletter, Stack Overflow's piece on shared guidelines, and the agentic pattern references, does not cite the empirical literature on identifier naming and model code comprehension in support of its naming recommendations. If someone shows me one that does, I will update this section.

That is a narrow claim, and I want it to stay narrow. I am not saying the existing guidance is wrong. "Use clear names" is correct advice and has been for fifty years. I am saying that in a period where three separate 2026 studies measured what specifically happens when names are stripped, misleading, or mutated, the advice has not been updated to reflect any of it, and it stops short of specifying a system. There is a difference between telling someone to eat well and handing them a diet with a shopping list. SSA tries to be the second thing: named layers, a naming grammar, machine-checkable invariants, and a skill that installs the workflow.

Borg and colleagues deserve a specific note, because their work is the closest empirical neighbor [19]. They measure structural code health and its relationship to AI refactoring outcomes across 5,000 Python files, and find a meaningful association. Their corpus is competitive-programming code rather than production systems, which limits how far it generalizes, but the direction is clear and it is measured. They do not examine identifiers. Their work and this one are looking at the same building from two sides.

## Convergent work

In April 2026, a month after the first version of this paper, two pieces arrived at conclusions close to mine, independently. Ustynov's work on semantic density and rethinking conventions for agentic development argues that verbose, meaning-dense identifiers are economically and cognitively correct for models, with the cost measurements discussed in Section 2 [12]. Tian Pan's "The AI-Legible Codebase" argues that the codebase itself, rather than the surrounding documentation, is the artifact that has to become readable to machines [20].

I found both after the first version of this paper was published. Three people reaching the same shape of answer is worth more than any one of us being first. The difference is scope. Both pieces advise. SSA specifies: five named layers, a grammar with an exact form, invariants a grep can check, an evolution story across two codebases, and a skill that sets it up in one conversation. Advice is easier to agree with. A specification is easier to test, and easier to prove wrong.

## Domain-driven design

The obvious ancestor is domain-driven design, and specifically the idea of a ubiquitous language: one shared vocabulary used by engineers and domain experts, in conversation and in code, so that the words in a meeting and the words in a class definition are the same words. SSA inherits that impulse. What it changes is the audience and the enforcement. The vocabulary now has a machine participant that reads it fresh every session, cannot ask a clarifying question in the hallway, and will confidently follow a misleading name straight off a cliff. That participant needs the vocabulary to be smaller, more consistent, and checkable.

## What SSA is not new about

Organizing directories so the tree announces the domain is screaming architecture, and Robert Martin wrote that in 2011. Package-by-feature is older. Past-tense event names come from event sourcing. The shared-vocabulary idea is domain-driven design, above. Boundary rules are enforced properly by dependency-cruiser and its relatives, not by my grep.

SSA does not claim any of those as inventions. What it claims is the assembly: an actor axis and a capability axis kept separate, one grammar spanning calls, errors, and events, a small verb set, invariants that either hold or do not, and a reason to bother that is measured rather than aesthetic. Every part has prior art. The reason to run them together arrived with the evidence in Section 2, and that evidence is two years old at most.

## One result worth keeping in view

In 2025, METR ran a randomized controlled trial with experienced open-source developers working in their own repositories. Those using AI assistance took 19% longer to complete their tasks, and believed they had been faster [21]. The trial did not test naming, and it did not test structure. It says nothing directly about SSA in either direction.

I include it because it is the most honest counterweight I know to the whole category of writing this paper belongs to. AI assistance is not automatically a speedup, and the people using it are poor judges of whether it is working. Any claim about a method that makes AI-assisted development better, including mine, should be read with that trial in the room.

# 6. Practice: Adopting SSA

This section is the practical guide. If you skipped here from the top, that was the right call.

Adopting SSA takes an afternoon for a small project. No rewrite and no migration plan. An afternoon of naming, and then a habit.

## Step one: pick your theme

Ask yourself one question. If my app were a place or a machine, what would it be?

Not a metaphor for what it does. A place you could walk around in. A ship. An observatory. A workshop. A city. A kitchen. An arcade. A garden. A heist crew. The answer does not need to be clever and it does not need to relate to your domain. A podcast host can be a radio station or a lighthouse. An expense splitter can be an inn. What you need is a world with enough parts in it that when you need a ninth name, the world hands you one.

Sit with it for ten minutes, pick the one that makes you smile, and move on. Ten minutes is enough here, and it should feel like naming a band rather than filling out a tax form.

## Step two: name three to five Systems

Not all of them. Three to five.

Open your code and ask what capabilities actually exist. Most small apps have four or five real ones under the noise: something that handles identity, something that handles money or the core transaction, something that stores the main thing users make, something that notifies, something that records. Name those. Draw the words from your theme. Write one plain sentence for each, in the form "Purse moves money between people."

Resist naming everything. Five words you use every day beat twenty you would have to look up. Systems six through twelve will announce themselves later, when you keep bumping into a responsibility that does not fit any existing word. That is the correct moment to add a name.

Then name your Surfaces, which is usually easier because there are two or three: the public thing, the signed-in thing, the admin thing, and an API if you have one.

## Step three: make the folders match

Create `lib/systems/` (or wherever code lives in your stack) with one folder per system, and put new code there from now on. Old code moves later or never. The vocabulary works in conversation before it works in the tree.

## Step four: write it down, briefly

Make a file called `CLAUDE.md` in your project's top folder. That is the file your agent reads at the start of every session. If your project already has an `AGENTS.md`, add to that instead. Same idea, different agents read it.

Put in it: a one-paragraph description of what the app is, a table of Systems with their one-line purposes, a list of Surfaces with who uses each, the grammar from Section 3, and the separation rules. Nothing else. The table is the part that does the work, and it looks like this:

| System | Purpose |
|---|---|
| Slate | The running record of who owes what |
| Purse | Money moving between people |
| Roster | Who lives here, who moved out, who is a guest |
| Bell | Notifications |

Aim for under 1,200 tokens, about a page of text. If your file is growing past that, the extra material is almost certainly describing something the code should be saying itself, and you should go fix the code instead of describing it better. That is the whole discipline in one sentence.

## The companion skill

I built a skill that does the above with you in one conversation. It is called `ssa-vocabulary-workshop`, and it is free and open source. Run this in your terminal, from inside your project folder:

```
npx skills add popdot-ai/ssa-vocabulary-workshop
```

It installs the workshop as a skill your agent can use. It does not touch your code. That command works across Claude Code, Cursor, Codex, and the other agents that support the Agent Skills format. Then start a session and say: "Help me set up SSA for this project."

The skill reads your project files and your top two directory levels, asks what theme you want, proposes three to five Systems based on what your code actually does, and names them with you, offering suggestions while you keep the final word. It then writes three markdown files: your context file, a `SYSTEMS.md` with the vocabulary, and a `NAVIGATOR.md` that maps common tasks to file paths.

Section 2 said context files do not measurably move correctness on their own. The skill writes three of them. That deserves an answer. The files are not the intervention. The naming conversation is, and the files are its transcript. A vocabulary table is one page, and it is the one thing an agent cannot infer from a codebase mid-rename. Once the tree matches the vocabulary, `SYSTEMS.md` should get shorter over time. If yours keeps growing, the code is not carrying its share, and the fix is in the code.

What it will not do matters more. It never renames your code and it never moves a file. Markdown is the only thing it writes, and if any of those three files already exists it will not overwrite it; it shows you the additions and asks. There are no scripts and no network calls in it, which is deliberate: independent security research from Snyk in February 2026 found that roughly a third of published agent skills carry a security finding [22]. The whole package is readable in five minutes, and you should read it, along with every other skill you install.

The skill is a convenience. Everything it does you can do by hand with the tables in Section 3, and doing it by hand is a perfectly good afternoon.

It is also deliberately thin, on purpose. In July 2026, members of the Claude Code team said Anthropic had removed roughly 80 percent of the Claude Code system prompt for its newest models, with older models keeping the full one, and described what they learned: prescriptive rules and "do not do this" instructions gave way to judgment, and worked examples turned out to constrain capable models rather than help them [23]. This skill applies that lesson. It carries no catalog of names to pick from. It states the quality bar, shows the register with a handful of examples, and lets the model generate candidates that fit your code. As models get smarter, the scaffolding around them thins. What remains is the artifact the model reads directly: your code, and what you named things. That part never gets cut.

## What changes the next day

The payoff shows up in what you say to your AI.

Before: three paragraphs explaining that there is a payments file, and another one that might be a duplicate, and a webhook handler somewhere that also touches money, and could it please look at all three and figure out why the refund is doubling.

After: "The bug is in Vault."

Other sentences you get for free: “Add this to Compass.” “Never let Market import Tower.” “This belongs in Slate, not Purse.” Each one replaces a paragraph, and more importantly each one is a sentence you can say with certainty, which is a different feeling from hoping the agent guesses right.

The same holds when you read. An error that says `Vault.Refund.InsufficientFunds` in your logs at 3am has already told you the system, the component, and the failure. A stack trace full of `processData` has told you to go read for twenty minutes.

## Growing, retiring, and merging

Vocabularies are living things. Three rules from five months of watching mine move.

**Add a name when you have felt the gap three times.** If you keep putting a piece of work somewhere that feels wrong, that is a System asking to exist. Name it from your theme, write its one-line purpose, add it to the table, and give it a directory the same day. A system that exists in your head but not in the file tree will not survive contact with an agent.

**Retire names cleanly.** When a capability goes away, take the name out of the vocabulary table and the directory out of the tree in the same change. On Popdot, a system named Gate (Content) was retired, and because it had a name and a home, removing it was bounded work rather than an archaeology project.

**Merge only when the user reality says so.** SAP merged an entire Surface, Helm, into Market, because owners and renters were the same people. The right trigger for a merge is discovering that two actors are one actor. The wrong trigger is that the codebase feels tidier with fewer folders.

## Adopting on a codebase you already have

You do not have to rename anything to start. Three passes, in this order.

First, name what exists. Read your directories, decide what the Systems already are, and write the vocabulary file. Even if not one identifier in your code matches it yet, you now have words, and you and your agent can use them in conversation immediately. “PaymentService is what we call Vault” works better than it has any right to.

Second, apply the grammar to new code only. Every new file goes in the right directory with the right name. This costs nothing extra and the proportion of legible code rises every week.

Third, rename opportunistically. When you are already in a file for another reason, bring it in line. Do not schedule a renaming sprint. A codebase where the parts you touch weekly follow the vocabulary is most of the benefit, and the untouched parts were not confusing anyone anyway.

The one exception worth doing deliberately: fix actively misleading names as soon as you find them. Section 2 is unambiguous on this point. A name that says the wrong thing is worse than a name that says nothing, and no amount of prompting or documentation repairs it.

## Objections, and working alone

Two people will disagree on a name. Decide who owns the vocabulary before the first workshop. Ties go to the shorter word, and once a name is in the tree it is final until the capability changes, not until someone thinks of a better one.

On a large codebase, an afternoon names the vocabulary and that is all it does. The words work in conversation immediately. The tree catches up over months, one opportunistic move at a time, and that is fine.

In a conservative shop, a heist crew will land badly, and the objection is not stupid. The fallback is to keep the layers, the grammar, and the rules, and pick sober words: Payments, Identity, Ledger. You lose the name generator and keep everything else.

If you work alone with no review process, the maintenance loop is one sentence you say once a week: "Check every file changed this week against SYSTEMS.md and list anything that violates the vocabulary or the import rules." The agent is good at exactly this kind of audit, and it keeps Wednesday's shortcuts from becoming next month's mystery.

## Staying in control

Here is the part I most want the person from Section 1 to hear.

More of your codebase is going to be AI-authored, not less. That is not a warning, it is just the trajectory. The question is what your relationship to that code is: owner, or tourist.

Names are how you stay the owner. You cannot read 40,000 lines a week, but you can hold a vocabulary of fifteen words, and if every one of those words maps to a place in the tree, you can find and check anything in under a minute. When an agent proposes a change, you can tell whether it belongs, and when something breaks you know where to look before you open a file. Hand the project to someone else, or come back to it in a year, and the map is still in the repository, because it is the repository.

You do not have to understand every line of a system to control it. You do have to be able to name every part.

# 7. Limitations

Two codebases. One developer. Me.

I designed the framework, applied it, and evaluated the result, which is three roles that are supposed to be held by different people. The productivity claims in this paper are self-reported estimates and should be read as such. I did not measure my sessions before adopting SSA and I cannot compare them honestly to my sessions after, because too much else changed in the same period, including the models.

There is a selection problem inside the evidence too. I chose the metaphors, so of course I find them memorable. A vocabulary that a stranger has to learn imposes a real cost that I never paid, and the only report I can give on that cost is from the two people who have asked me questions about my code. That is not data.

Both platforms are marketplaces. Both were built by one person with an AI agent. SSA has not been tested on a team, on a monolith with fifteen years of history, on embedded systems, on data pipelines, or anywhere the actor model of Surfaces might not fit. I suspect Surfaces get less useful as the number of distinct actors approaches one. I have not verified that.

The studies in Section 2 support the premises, not the conclusion. Nobody has tested Semantic Surface Architecture as a system.

Here is what a real test would look like, described concretely enough that someone could run it. Take one repository. Produce two versions that differ only in identifier and directory naming, one following SSA and one following conventional service and utility naming, with logic held identical. Give both to several agents from different model families. Run matched tasks: fix a seeded bug, add a feature that spans two capabilities, answer a question about where a responsibility lives. Measure exploration steps before the first edit, first-try correctness, edits made to the wrong file, and total session cost. Jin's methodology already measures exploration steps in a comparable way [13], and Haroon's mutation approach already shows how to hold logic constant while changing names [7]. The pieces exist.

My prediction, recorded here so it can be checked against me: SSA would show a clear advantage on exploration steps and wrong-file edits, a smaller advantage on first-try correctness, and no advantage at all on tasks confined to a single file. If someone runs it and finds nothing, I would want to know, and I would rather find out from a study than from a codebase.

# 8. An Invitation

If you are six weeks into something you no longer recognize, do not start over. Open the repository, look at what is actually in it, and name five things. Give them a world to come from. Write the page. Then tell your agent where the bug is instead of hoping it finds out.

That is the whole practice. It costs an afternoon and it does not require permission from anyone.

I would like to hear where it breaks. Which parts of this held up in a domain I have never worked in, which rules you had to bend, what you named your systems, and what you were forced to call the thing that would not fit in one word. I have two data points and a framework that deserves harder ones. Send me yours.

# References


[1] Paipuru, T. (2026). The Navigation Paradox in Large-Context Agentic Coding: Graph-Structured Dependency Navigation Outperforms Retrieval in Architecture-Heavy Tasks. arXiv:2602.20048.

[2] Chroma (2025). Context Rot: How Increasing Input Tokens Impacts LLM Performance. Chroma Research.

[3] Wang, Z., Zhang, L., Cao, C., Luo, N., Luo, X., Liu, P. (2023, revised 2024). How Does Naming Affect LLMs on Code Analysis Tasks? arXiv:2307.12488.

[4] Le, C.C., Pham, M.V.T., Van, C.D., Phan, Hoang N., Phan, Huy N., Nguyen, T.N. (2025). When Names Disappear: Revealing What LLMs Actually Understand About Code. arXiv:2510.03178.

[5] Le, J., Nguyen, A.H.N., Nguyen, T.N. (2026). Do Machines Struggle Where Humans Do? LLM and Human Comprehension of Obfuscated Code. arXiv:2606.31725.

[6] Guzmán Lorenzo, L. (2026). Poisoned Identifiers Survive LLM Deobfuscation: A Case Study on Claude Opus 4.6. arXiv:2604.04289.

[7] Haroon, et al. (2025). Assessing the Impact of Code Changes on the Fault Localizability of Large Language Models. ICST 2026. arXiv:2504.04372.

[8] Khatri, P. (2026). Do Context Files Help Coding Agents? A Two-Agent Ablation Study on Real Repositories. arXiv:2607.27250.

[9] Chatlatanagulchai, et al. (2025). Agent READMEs: An Empirical Study of Context Files for Agentic Coding. arXiv:2511.12884.

[10] Vasilopoulos, A. (2026). Codified Context: Infrastructure for AI Agents in a Complex Codebase. arXiv:2602.20478.

[11] Anthropic (2025). Effective context engineering for AI agents. Anthropic Engineering, September 29, 2025.

[12] Ustynov, D. (2026). Beyond Human-Readable: Rethinking Software Engineering Conventions for the Agentic Development Era. arXiv:2604.07502.

[13] Jin, R. (2026). Formal Architecture Descriptors as Navigation Primitives for AI Coding Agents. arXiv:2604.13108.

[14] Claude Code documentation. Skills. code.claude.com/docs/en/skills.

[15] Agent Skills open standard. agentskills.io.

[16] Nous Research (2026). Hermes Agent documentation. hermes-agent.nousresearch.com/docs.

[17] Shapira, et al. (2026). Agents of Chaos. arXiv:2602.20021.

[18] AGENTS.md. Open format for agent instruction files, governed by the Agentic AI Foundation (Linux Foundation) since December 2025. agents.md. See also Linux Foundation press release, December 9, 2025, linuxfoundation.org.

[19] Borg, M., Hagatulah, N., Tornhill, A., Söderberg, E. (2026). Code for Machines, Not Just Humans: Quantifying AI-Friendliness with Code Health Metrics. arXiv:2601.02200.

[20] Pan, T. (2026). The AI-Legible Codebase. tianpan.co, April 2026.

[21] METR (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR. arXiv:2507.09089.

[22] Snyk (2026). ToxicSkills: a study of the agent skills supply chain. Snyk Research, February 5, 2026.

[23] Shihipar, T. and Cat (Anthropic Claude Code team) (2026). Fireside chat, AI Engineer World's Fair, July 21, 2026. See T. Shihipar, x.com/trq212/status/2080710971228918066, and coverage at simonwillison.net/2026/Jul/21/cat-and-thariq/.

# Appendix: The full vocabularies

**Popdot AI, 22 Systems.**

| System | What it does |
|---|---|
| Vault | Money movement |
| Sigil | Agent identity |
| Lease | The rental lifecycle |
| Relay | Subdomain routing and serving |
| Drop | Content deployment |
| Tether | DNS records |
| Latch | Domain connections |
| Prism | Pricing |
| Orbit | Search |
| Echo | Agent discovery and marketing |
| Signal | Real-time events |
| Beacon | Analytics |
| Bloom | Email and agent webhooks |
| Pulse | Background jobs |
| Gate | Human authentication |
| Trial | Free trials |
| Security (5 systems) | Screening and abuse prevention. Names withheld. |
| (unnamed) | Admin plumbing under Cortex |

**Sleep Around Points, 19 Systems.**

| System | What it does |
|---|---|
| Gate | Auth and identity verification |
| Deed | DVC contracts, a real-estate metaphor |
| Shelf | Point listings |
| Orbit | The trip request board, where renters post what they want |
| Ledger | The booking lifecycle |
| Vault | Money movement |
| Prism | Pricing and point charts |
| Atlas | Resort data |
| Signal | Messaging |
| Shield | Privacy and abuse controls |
| Arbiter | Disputes |
| Crank | Scheduled jobs |
| Ping | Email |
| Locker | File storage |
| Scribe | Audit logging |
| Pact | Rental agreements and e-signing |
| Compass | First-party analytics |
| Beacon | Conversion events |
| CAPI | Third-party conversion tracking |

END OF PAPER

## About the author

Michael Latulippe builds production software with AI agents as collaborators. Semantic Surface Architecture came out of that work on two platforms described in this paper: Popdot AI, an agentic domain rental marketplace, and Sleep Around Points, a Disney Vacation Club points rental marketplace. He lives in Celebration, Florida.

This work is offered for discussion and experimentation. Correspondence, criticism, and collaboration are welcome at hello@michaellatulippe.com.

## How to cite



PLAIN CITATION

Latulippe, M. (2026). *Semantic Surface Architecture: A naming system for codebases built with AI* (Version 2.0). www.michaellatulippe.com/whitepaper/semantic-surface-architecture

BIBTEX

```
@techreport{latulippe2026ssa,
  author  = {Latulippe, Michael},
  title   = {Semantic Surface Architecture},
  version = {2.0},
  year    = {2026},
  month   = {8},
  note    = {CC BY 4.0. First published March 2026},
  url     = {https://www.michaellatulippe.com/whitepaper/semantic-surface-architecture}
}
```