ChatGPT

Common Architecture Language Model (CALM) and Reverse-Engineering System Architecture

What is CALM?

The **Common Architecture Language Model (CALM)** is an open-source specification under FINOS's "Architecture-as-Code" initiative. It lets software architects **define, validate, and visualize system architectures in a standardized, machine-readable format** [1]. In CALM, architecture is expressed as JSON following a common schema, bridging the gap between high-level design and actual implementation [1]. CALM treats architecture as versionable code, much like Infrastructure-as-Code, enabling architects to keep design documentation in sync with evolving systems.

Key elements of CALM's model include **Nodes**, **Relationships**, and associated metadata [2]. Nodes represent components of the system (e.g. services, databases, UIs, networks), while Relationships capture how those components interact (calls, data flows, dependencies). Metadata and controls can be attached for additional context (like ownership, environment, or compliance requirements) [2]. This modular approach allows modeling anything from a high-level system overview down to detailed microservice architectures [2]. CALM focuses on architecture rather than code internals, making it **language-agnostic** – you can describe systems built in Python, Java, JavaScript/TypeScript, etc. using the same CALM schema.

Reverse-Engineering Architecture from Source Code Using CALM

Does CALM support reverse-engineering architecture from code? Yes – one of CALM's use cases is capturing the architecture of existing applications (even if they weren't originally modeled in CALM). In fact, CALM is *already being used in industry to document the architectures of existing systems* [3]. This means architects can take an existing codebase and derive a CALM model that represents its structure and interactions. While CALM doesn't automatically parse source code out-of-the-box, the **Architecture-as-Code toolkit and community provide tools and patterns** to assist with extracting architecture from code and configuration.

Approach to Extracting Architecture from Code

1. Identify Architectural Components in Code: Start by analyzing the codebase (manually or with tools) to find the major architectural elements: - **Services or Modules:** For example, each microservice (such as a Python Flask app, a Java Spring Boot service, a Node/TypeScript Express service, etc.) could be a Node in CALM. In a monolithic application, key layers or subsystems might be treated as Nodes. - **Datastores and External Systems:** Identify databases, caches, message brokers, third-party APIs, etc. These become Nodes (often of type *Database* or *ExternalSystem* in the CALM model). - **Communication Paths:** Determine how components interact – e.g. REST calls, gRPC calls, function calls across modules, messaging topics, or database access. These will be modeled as Relationships (with Interfaces detailing the protocol or data format). For instance, "Service A calls Service B's REST API" would be a Relationship with an HTTP interface

defined. - **Infrastructure or Network Boundaries:** If relevant, note network zones, load balancers, or client applications. These can also be Nodes or captured via metadata.

2. Represent Components as CALM Nodes: Once you have the list of components from the code: - Create a CALM JSON **architecture file** (by convention often named `architecture.json` or `*.calm.json`). Each component is listed as a Node object, with attributes like `name`, `type` (service, database, UI, etc.), and any relevant interfaces it provides. For example, a Java service might be a Node of type "ApplicationService" with an interface for an HTTP endpoint. - You can do this manually or use **generation tools/patterns**. CALM comes with **predefined architecture patterns** that can be used as templates. For instance, CALM provides a *microservices pattern* JSON template. Using the CALM CLI's generate command, you can scaffold a new architecture from such a pattern and then fill in details. *E.g.*: `calm generate -p calm/pattern/microservices.json -o my-architecture.json` will create a JSON skeleton for a microservices architecture [4], which you can edit to match your system.

3. Add Relationships and Interfaces: Next, for each interaction identified: - Add a **Relationship** entry in the CALM model connecting the source and target Nodes. For example, if a Python API service reads from a PostgreSQL database, represent that with a relationship from the API Node to the DB Node. - Define the **Interface** or communication method. CALM uses interface objects (often referenced in Nodes and Relationships) to specify protocols or data formats. In the above example, the interface might be of type "JDBC" or "Postgres connection" on the database side, or an HTTP REST interface between two services. CALM enforces certain schema rules here (for instance, an interface may require specifying one of a set of allowed types) – the CLI's validation can help ensure correctness.

4. Include Metadata and Controls (Optional): To enrich the reverse-engineered model, you can add metadata like the owner team, deployment environment, or criticality of each component [5]. CALM also supports **Architecture Controls** – basically policy or compliance requirements (security controls, standards) that certain nodes or relationships should adhere to [6]. For example, if the codebase uses encryption or follows specific compliance (PCI, FIDO2, etc.), you can encode those as controls in the CALM model for auditing. This step ties into CALM's strength in automated governance – once the model is extracted, CALM can validate if the architecture meets all required controls.

5. Validate and Iterate: With an initial CALM JSON model drafted, use the **CALM CLI** to validate it and refine: - Run `calm validate -a my-architecture.json` to check the model against the CALM schema and any pattern you want to enforce. The CLI will report errors if, say, required fields are missing or relationships don't meet schema rules [7] [8]. This helps catch mistakes in the reverse-engineered model. - Iterate on the model: it's common to go back to the code or system docs to verify components and add missing pieces. This process can be collaborative – because the architecture is code, you can commit the `architecture.json` to version control and review it just like any source code change.

6. Visualize and Document: Once the model aligns with the system: - **Live Visualization:** Use CALM's VS Code extension "*CALM Preview & Tools*" to live-visualize the architecture as you edit the JSON [9]. The extension provides an interactive diagram view that updates in real-time, showing nodes and relationships. This is invaluable for validating that the reverse-engineered architecture "looks" right. You can click on elements in the diagram to inspect their details [10]. The extension also offers a tree navigation of the architecture model and will highlight any validation errors inline [11]. - **Documentation Generation:** CALM can auto-generate documentation from the model. The CLI's `calm docify` command can produce a website or markdown documents from the CALM JSON (leveraging templates) [12]. For example, after

reverse-engineering a Java system's architecture, you could run `calm docify -a my-architecture.json -o docs/` and get a site with component tables, relationship diagrams, and diagrams (block diagrams, sequence flows, etc.) derived from the model. This ensures your "Solution Architecture Description" stays in sync – any code changes can be reflected by updating the CALM model and regenerating the docs [13] [14] . - **CALM Hub:** If you want to share or curate architecture models, FINOS provides **CALM Hub**, a central repository for CALM artifacts (patterns, architectures, controls) [15] . You can publish your extracted model to CALM Hub for collaboration or reuse. For example, a common microservice architecture extracted from a reference project could be stored as a pattern for others to leverage.

Leveraging AI and Plugins for Automation

Manually analyzing a large codebase can be time-consuming. CALM's ecosystem includes **AI-assisted tools** to automate architecture extraction to a large extent: - **CALM Copilot Chatmode:** CALM offers deep integration with GitHub Copilot (ChatGPT-based assistant in VS Code) to help generate and modify architectures. By running `calm copilot-chatmode` in your project, CALM sets up a specialized *chat configuration* for Copilot [16] . This provides the AI with knowledge of the CALM schema, validation rules, and best practices [17] [18] . In practice, this means you can **ask Copilot to create an architecture model from descriptions or code context**. For example, you might supply Copilot with a high-level description of your system (or even snippets of your codebase) and prompt it to "Generate a CALM architecture JSON for this application." In a FINOS demo, Morgan Stanley engineers showed that they could build a complex trading system architecture in under 10 minutes by *"talking to the architecture"* via the VS Code Copilot chat – without writing JSON by hand [19] [20] . The AI can add nodes, relationships, and even rename or reorganize components on command [21] [22] . This dramatically speeds up reverse-engineering: the architect guides the AI, which writes the CALM model.

How it works: The `.github/chatmodes/CALM.chatmode.md` config injected by `calm copilot-chatmode` teaches Copilot about the CALM JSON structure and includes **tooling prompts** for each modeling task (creating nodes, relationships, interfaces, controls, etc.) [23] [24] . With this in place, you can ask, for example, *"Copilot, create nodes for each microservice in my repo and link them"*. Copilot will output JSON fragments following the CALM schema (for example, a Node list with names of services and a relationship matrix) which you can then refine. Essentially, **Copilot becomes an architecture reverse-engineering assistant** – it won't magically know everything about your code, but if you describe the code structure or provide file names, it can infer a reasonable model. This approach is aided by the fact that CALM's schema is strict; Copilot suggestions will align with the JSON Meta-Schema definitions of CALM, reducing syntax errors [25] .

Natural Language Example: A Medium tutorial on CALM showed using Copilot to generate a Passkey/WebAuthn authentication architecture. The author provided a prompt file with system details, and Copilot produced an `authn.calm.json` model capturing nodes for the Auth Service and WebAuthn Client, their relationship, and security controls [26] . This illustrates how you can feed **instructions or even parsed code information** to Copilot to yield a CALM model. In essence, you describe "what the code does" and the AI outputs the structured architecture. This can be done incrementally: e.g., "List all services in this repository as CALM nodes" then "For each service, add its database as a node and a relationship." Each prompt coaxes the architecture out of the code.

- **Plugins & Custom Tools:** The Architecture-as-Code project is designed to be extensible. The FINOS CALM monorepo has a `calm-plugins` directory [27] and community encouragement to develop

new capabilities that use the CALM spec. While specific language-parsing plugins may still be in early stages, one can imagine (or build) plugins for static code analysis:

- *For Java:* A plugin could scan a Maven project for `@SpringBootApplication` classes or microservice modules, then output CALM JSON Nodes for each, and use frameworks like ArchUnit or jQAssistant to find inter-service calls (REST client usage, message queues) to form Relationships.
- *For Python:* A script might inspect a Django or Flask project. For example, it could list Django apps or Flask Blueprint registrations as Nodes, detect database configurations (settings.py) as a DB Node, and link them. Similarly, parsing import statements or function calls could uncover dependencies.
- *For JavaScript/TypeScript:* One could parse an Express.js or NestJS app to identify distinct modules or services. For front-end interactions, perhaps automatically include a Node for a React frontend that communicates with a backend API Node via a REST interface. Static analysis of import graphs or API route definitions could inform these relationships.
- *For any language:* If the system is deployed with infrastructure-as-code (Docker Compose, Kubernetes manifests, Terraform, etc.), those definitions can be mined. E.g., a Docker Compose file lists services and networks – a tool could transform that into CALM Nodes and network Relationships. In this way, CALM can *ingest architecture from deployment descriptors* as well as code.

While these specific plugins may not be officially released yet, the **CALM community is actively exploring automation**. The CALM AI project and community demos indicate a trend toward minimal manual effort ("zero JSON") in creating architecture-as-code [20] [22] . In other words, the tools aim to *reverse-engineer and maintain architecture models automatically* using code analysis and AI. As CALM matures, we can expect more out-of-the-box support for scanning source code and generating initial CALM models.

Example Workflow by Language

Because CALM's format is technology-neutral, you follow a similar workflow for different tech stacks, with adjustments for each:

- **Python:** Suppose you have a Django web application with a PostgreSQL database. You would create Nodes for the Django app, the Postgres DB, and perhaps an external API it calls. Using Python introspection, you might list Django apps (each could be a subsystem Node) or analyze the Django URL routes to identify external interfaces. Each model (e.g., a Django app handling authentication) might become a Node with an interface (e.g., "HTTP REST API /auth/*"). The database is a Node of type "Database" with an interface like "PostgreSQL connection". A Relationship is added from the Django Node to the DB Node representing the ORM calls. After extracting these from code (Django settings for the DB name, urls.py for endpoints, etc.), you'd encode them in JSON and validate with CALM. The CALM CLI and Copilot** can then be used to fill in details or enforce that required security controls (maybe you require TLS on all DB connections – a control can ensure that).

- **Java (Spring Boot Microservices):** If you have multiple Spring Boot services communicating via REST and Kafka, each service is a CALM Node (type "Service" or more specifically "SpringService"). Interfaces could be HTTP (for REST endpoints on each) and a Kafka interface for messaging. You might use a static analysis tool or even runtime info (e.g., reading Spring config files or API docs) to list all endpoints and topics. These become relationships: e.g., *OrderService -> PaymentService* (HTTP REST call for charging) and *OrderService -> Kafka: OrderEvents* (a relationship from OrderService to a Kafka "node" or a pseudo-node representing the event bus). CALM can capture both synchronous and asynchronous flows. You'd add metadata like `language: Java` or `framework: Spring` as

metadata if desired (though not required). With CALM's pattern capability, you could start from a built-in **"microservice pattern"**, which may include placeholders for service nodes and common relationships, then map your specific code components onto it [28] [4]. Finally, use the VS Code extension to visualize that all microservices and links are correctly represented (the live diagram would show each service box and arrows for calls) [10].

- **TypeScript/JavaScript:** Consider a Node.js backend with a React frontend. You'd have a Node for the Node.js API server, Nodes for any external services it uses (e.g., a MongoDB, or third-party REST API), and a Node for the React single-page app (as a client component). From code, you identify API endpoints in the Node.js server (perhaps via an Express route list) – these form an interface (REST/HTTP) on the Node.js Node. The React app Node might have an interface representing the browser UI. The relationship: React Node -> Node.js API Node (HTTP calls), Node.js Node -> MongoDB Node (database queries), etc. Since JavaScript/TypeScript are dynamic, you might rely on documentation or runtime logging to extract these interactions. Tools like dependency graph generators (Webpack or AST analyzers) could help find imports and usage. Once listed, define them in CALM JSON. Because CALM doesn't limit what tech is used, you can mark a Node's metadata with "runtime: Node.js" or similar for clarity, but structurally it's no different than other services. After modeling, you can validate the model and use `calm docify` to produce, say, a Markdown that lists all API endpoints and which component provides each – useful for onboarding new developers to the codebase.

- **Multi-Language System:** If your application spans multiple languages (say a Java backend, a Python machine learning service, and a JavaScript front-end), CALM can **unify the architecture description**. Each part is a Node, and CALM doesn't require separate models per language. This is especially powerful for reverse-engineering enterprise systems where different components are written in different stacks. You can extract each piece's info in its native way, then combine them into one CALM architecture file. The CALM CLI's validation will ensure consistency (for example, you can't accidentally link a relationship to a non-existent node – all references must match). The end result is a *single architecture view of the whole application*, which you can visualize or export. This holistic model can be kept alongside the code (e.g., in the repository root), so anyone navigating the repo can refer to `architecture.json` to quickly grasp the system's structure.

Tools, Plugins, and Examples

To summarize the key tools and steps available for extracting architecture with CALM:

- **CALM CLI (`@finos/calm-cli`)** – The core command-line tool to generate, validate, and document architectures. You can install it via npm and use commands like `calm generate` (to scaffold from a pattern) and `calm validate` (to check your model) [29] [7]. The CLI is language-neutral; it operates on the CALM JSON. For example, after reverse-engineering a system, run `calm validate -p some-pattern.json -a my-architecture.json` to ensure your extracted model conforms to an expected pattern [7]. You can also use `calm validate -f junit` to produce output consumable by CI pipelines [30] – i.e., you can fail a build if the architecture model doesn't meet certain rules (useful when you want to enforce that the code and architecture stay in sync as code evolves).

- **VS Code Extension (CALM Preview & Tools)** – A Visual Studio Code plugin published by FINOS (available on the VSCode Marketplace) which provides an **interactive preview of CALM models** [9]. When reverse-engineering, this is extremely helpful: as you add nodes/relations to the JSON file, you instantly see the diagram update. The extension also offers intelligent validation (showing schema errors in the editor Problems pane) and a tree view for navigating large architectures [31] [32]. It effectively turns VS Code into a lightweight architecture modeling IDE. As of v0.1.0 (Dec 2025) the extension graduated from experimental to active status, indicating it's stable and feature-rich [33]. For instance, you can toggle between a *diagram view* and a *documentation view* – the latter can generate markdown/HTML docs from the model in real time [34]. This extension supports any CALM JSON, regardless of the underlying application language (the content is the architecture).

- **CALM Hub** – A repository (and likely web UI) for storing and sharing architecture models and patterns. While not directly extracting from code, CALM Hub can host *reference architectures* and *reusable patterns*. For example, FINOS provides patterns for common architectures (API gateway setups, microservices, data pipelines, etc.). These can guide your reverse-engineering: you might find a pattern that resembles your system and use it as a starting point. CALM Hub also allows organizations to maintain a catalog of their approved architectures and **compare extracted models against approved patterns**. According to FINOS, CALM Hub is part of the tooling released with CALM v1.0 [15].

- **Community Plugins and Integrations:** The CALM project is open-source and evolving. Community contributors are working on various integration points. Some notable ones:

- **CALM Agents/AI**: Morgan Stanley developed an internal "CALM Agent" (AI-driven assistant) demonstrated at OSFF (Open Source Finance Forum) 2025, which could *generate an architecture as code without writing JSON* [19] [20]. This agent likely builds on the Copilot Chatmode discussed above, and showcases how AI can drastically speed up the architecture extraction and maintenance process. While the agent itself is basically the Copilot integration, it highlights a direction where you might simply point the agent at a repository and ask for an architecture, and get a first draft of a CALM model.
- **SonarQube Integration**: Though not officially part of CALM, the concept of "architecture as code" has been recognized by static analysis vendors (e.g., SonarQube's recent feature to analyze code structure for architecture rules). It's conceivable to integrate Sonar or other static analyzers with CALM – for example, a Sonar plugin could export detected module dependencies in CALM format, or at least validate that the code's dependencies don't violate the CALM model (catching architecture drift). This isn't provided out-of-box by CALM, but demonstrates that CALM can serve as a **formal spec to compare against the code**.

- **Language-Specific Tools**: We mentioned ArchUnit (for Java) and similar tools. A workflow could be: write ArchUnit tests that ensure the code's package structure or call graph matches your CALM architecture. In fact, because CALM is JSON, you could generate a list of allowed dependencies from the CALM file and use a static analysis tool to verify the code doesn't introduce others (for instance, if CALM says Service A talks only to Service B, you'd flag if Service A suddenly makes a call to Service C in code). This kind of **round-trip integration** (architecture <-> code) is the long-term benefit of using CALM.

- **Example Configurations:** An example CALM architecture JSON snippet (from a generic microservice system) might look like:

```
{
  "nodes": [
    { "name": "OrderService", "type": "Service", "interfaces": [{ "$ref":
"#/interfaces/OrderAPI" }] },
    { "name": "PaymentService", "type": "Service", "interfaces": [{
"$ref": "#/interfaces/PaymentAPI" }] },
    { "name": "OrderDB", "type": "Database", "interfaces": [{ "$ref": "#/
interfaces/Postgres" }] }
  ],
  "relationships": [
    { "source": "OrderService", "target": "PaymentService", "interface": {
"$ref": "#/interfaces/REST" } },
    { "source": "OrderService", "target": "OrderDB", "interface": {
"$ref": "#/interfaces/JDBC" } }
  ],
  "interfaces": {
    "OrderAPI": { "protocol": "HTTP", "address": "/order/*" },
    "PaymentAPI": { "protocol": "HTTP", "address": "/pay/*" },
    "REST": { "protocol": "HTTP", "description": "REST call" },
    "Postgres": { "protocol": "JDBC", "description": "PostgreSQL DB
connection" },
    "JDBC": { "protocol": "JDBC" }
  },
  "metadata": { "systemName": "E-Commerce Platform", "owner": "Team
Checkout" }
}
```

This is illustrative – it shows how a CALM model might encode services and a database with their relations. In practice, the actual schema has specific structure (e.g., `interfaces` might be an array rather than an object map). The CALM JSON **schema** is available in the CALM GitHub (and referenced via `$ref` in patterns) [35] [27]. You would ensure your generated JSON conforms to it. Using **CALM's JSON meta-schema** means any standard JSON Schema tools or validators can also be used to validate the architecture files.

Conclusion

CALM can indeed be used to reverse-engineer system architecture from source code, with support for popular languages like Python, TypeScript/JavaScript, Java, and more. Its strength lies in providing a uniform language to describe architecture, separate from the source code languages. The process involves analyzing the code to identify components and interactions, encoding them in CALM's JSON format, and using CALM's tooling to validate and visualize the model. While CALM doesn't yet auto-generate architecture

from code at the push of a button, the combination of **static analysis, patterns, and AI-assisted tools (Copilot integration)** makes the task much easier:

- **Python, JavaScript/TypeScript, Java, etc.** – all can have their architectures modeled in CALM. The differences lie in how you extract the info (using language-specific parsers or frameworks), but once in CALM, the models are handled uniformly. CALM abstracts away the code specifics and focuses on *architectural building blocks* (services, databases, interfaces), which is why it's applicable across languages [2].
- **Tools and examples:** FINOS provides the CALM CLI and VSCode plugin to work with these models, and early adopters have demonstrated using CALM to capture architectures of legacy systems and even automate architecture documentation for large applications [3] [36]. The CALM website and GitHub have examples and patterns that you can study – for instance, sample CALM files for an "API Gateway + Microservices" pattern, etc., which can guide your reverse-engineering effort [4].

By linking architecture and code, CALM allows you to continuously maintain the architecture model as the code evolves. Reverse-engineering is not a one-time event – with CALM, you can set up a pipeline (even a **CI check**) so that whenever the code changes in a way that violates the documented architecture, you get alerted [37]. This closes the loop between code and design. In summary, **CALM supports reverse-engineering architectures** through a combination of its schema (to represent any system's structure), its tooling (for visualization, validation, and documentation), and integrations (AI assistance and potential static analysis plugins). It brings the practice of software *architecture-as-code* to reality – turning the architecture of your Python, TypeScript, Java, or JavaScript application into a living artifact that can be generated from, checked against, and evolved alongside the source code.

Sources

- CALM official documentation – *FINOS Architecture-as-Code*: Introduction to CALM and core concepts [1] [2], usage of CALM CLI (generate/validate) [4] [7], and CALM Copilot integration [16] [18].
- FINOS Community announcement of CALM v1.0: notes that CALM is used to *"document architectures of existing systems"* and mentions tools like CALM Hub and a VS Code plugin [15] [36].
- *DevOps.com* news article on CALM's open-sourcing by Morgan Stanley: confirms industry usage of CALM for existing architectures and describes CALM's JSON model (nodes, relationships, etc.) [2] [3].
- CALM VS Code Extension (Marketplace) – details features for live visualization, tree navigation, and documentation generation in-editor [9] [38].
- *Medium* article by K.V. Perumal (Dec 2025): *"Modeling Passkey/WebAuthn with CALM"* – demonstrates using GitHub Copilot to generate CALM JSON from natural language and how CALM ensures alignment between implementation and architecture [39] [26]. This highlights the AI-assisted reverse-engineering approach (though applied to a specific use-case).
- FINOS community blog *"Confessions of a CALM-aholic"* – recap of a live demo where an architect used the CALM AI Copilot to build a complex system architecture without manually writing JSON, indicating how CALM's AI tooling can automate architecture creation [19] [20].

---

[1] [2] [3] Morgan Stanley Open Sources CALM: The Architecture as Code Solution Transforming Enterprise DevOps - DevOps.com
https://devops.com/morgan-stanley-open-sources-calm-the-architecture-as-code-solution-transforming-enterprise-devops/

4 28 29 Generate | Architecture as Code
https://calm.finos.org/working-with-calm/generate/

5 6 12 26 37 39 Modeling Passkey/WebAuthn with CALM: Architecture as Code in VS Code with GitHub Copilot | by KV Perumal | Dec, 2025 | Medium
https://indianakv.medium.com/modeling-passkey-webauthn-with-calm-architecture-as-code-in-vs-code-with-github-copilot-9b16d1d3f576

7 8 30 Validate | Architecture as Code
https://calm.finos.org/working-with-calm/validate/

9 10 11 31 32 33 34 38 CALM Preview & Tools - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=FINOS.calm-vscode-plugin

13 14 19 20 21 22 Confessions of a "CALM-aholic": How Morgan Stanley Automates Architecture Without Writing JSON
https://www.finos.org/blog/confessions-of-a-calm-aholic-how-morgan-stanley-automates-architecture-without-writing-json

15 36 [FINOS Community] CALM v1.0 Released
https://groups.google.com/a/finos.org/g/odp/c/E60cEv76-rg

16 17 18 23 24 25 CALM Copilot Chatmode | Architecture as Code
https://calm.finos.org/working-with-calm/copilot-chatmode/

27 GitHub - finos/architecture-as-code: "Architecture as Code" (AasC) aims to devise and manage software architecture via a machine readable and version-controlled codebase, fostering a robust understanding, efficient development, and seamless maintenance of complex software architectures
https://github.com/finos/architecture-as-code

35 Welcome to CALM | Architecture as Code
https://calm.finos.org/