Constructing the Aggregated Interface Map: A Comprehensive Workflow without Abstract Syntax Trees

Section 1: Defining the Aggregated Interface Map (AIM)

1.1 Deconstructing the Concept: A Synthesis of Three Pillars

The modern enterprise software landscape is characterized by a sprawling network of interconnected services, libraries, and platforms. Understanding this complexity is no longer a matter of simple dependency graphing; it requires a holistic architectural model that bridges business context with technical reality. The "Aggregated Interface Map" (AIM) is proposed as such a model—a multi-layered, dynamic representation of a software system's architectural dependencies and communication patterns. It is not merely a static diagram of code connections but a living model of the system's socio-technical structure.

The definition and purpose of the AIM can be deconstructed from its constituent terms, which synthesize concepts from domain modeling, interface design, and systems analysis. These three pillars form the conceptual foundation of the AIM:

1. **"Aggregated":** This term refers to the logical grouping of software components into cohesive, self-contained units with clear consistency boundaries. This concept is drawn directly from the strategic patterns of Domain-Driven Design (DDD), where an "Aggregate" is a cluster of related objects that are treated as a single unit for data changes.[1] The AIM operates at this level of abstraction, mapping the relationships between these business-aligned aggregates rather than focusing on the granular, and often noisy, connections between individual classes or functions. This approach elevates the map from a low-level implementation detail to a high-level architectural blueprint that

reflects the structure of the business domain itself.[3]

2. **"Interface":** This pillar defines the "how" of inter-aggregate communication. An interface represents the formal, explicit contract through which an aggregate exposes its capabilities to the outside world.[5] In the context of the AIM, interfaces include not only formal API definitions (such as those described by OpenAPI specifications) but also the data structures, message schemas, and communication protocols that govern interactions.[7] The AIM is fundamentally concerned with mapping the dependencies between these well-defined boundaries, enforcing the principle that external components should only interact with an aggregate through its designated public interface, known as the "Aggregate Root".[1]
3. **"Map":** This signifies the visual and analytical representation of the relationships discovered between the aggregated interfaces. It is a dependency graph, but one that is significantly enriched with multiple layers of information. The map visualizes both the *static*, potential dependencies declared in code and configuration, and the *dynamic*, actual dependencies observed in a running system.[8] This dual perspective allows for the identification of architectural drift, performance bottlenecks, and unused components, making the map a powerful tool for analysis and decision-making.[9]

The synthesis of these three pillars yields a powerful conclusion about the nature of the AIM. It is an artifact designed to bridge the persistent gap between business architecture and technical implementation. By grounding the dependency map in the language and structure of the business domain (via DDD aggregates), the AIM provides a common language for architects, developers, and business stakeholders. It moves beyond answering the simple question, "What is connected to what?" to address the far more strategic question: "How do our core business capabilities, as embodied in software, actually interact and depend on one another?" This framing makes the AIM an indispensable tool for analyzing the alignment between technical coupling and business domain cohesion.

1.2 The Role of the DDD Aggregate as an Architectural Quantum

To effectively manage the overwhelming complexity of modern systems, it is necessary to identify a fundamental unit of analysis—an "architectural quantum" that is large enough to be meaningful from a business perspective but small enough to be technically manageable. The Aggregate pattern from Domain-Driven Design provides the ideal candidate for this role within the AIM framework.

An Aggregate is a collection of one or more related entities and value objects that are treated as a single, cohesive unit.[1] Each aggregate is governed by a single entity known as the "Aggregate Root," which serves as the exclusive entry point for all commands and queries

directed at the aggregate's members.[2] This design pattern is not merely an organizational convenience; it is a mechanism for enforcing transactional consistency and business invariants. Any operation that modifies the state of the aggregate is executed atomically, ensuring that the entire cluster of objects remains in a valid state at all times.[3]

By adopting the aggregate as the primary node in the AIM, the map's focus shifts from implementation minutiae to architectural significance. Instead of a tangled web of class-to-class dependencies, the AIM presents a clear view of the interactions between major business capabilities. For example, in an e-commerce system, instead of mapping the hundreds of dependencies between OrderItem, ProductDetails, and ShippingAddress classes, the AIM would represent the entire concept as a single "Order" aggregate. The map would then visualize how this "Order" aggregate interacts with other aggregates like "Customer," "Inventory," and "Payment."

This abstraction is critical for several reasons:

- **Complexity Management:** It drastically reduces the number of nodes and edges in the dependency graph, making the overall architecture comprehensible to human stakeholders.[2]
- **Alignment with Business Logic:** It ensures that the architectural diagram reflects the structure of the business domain, facilitating communication and alignment between technical and non-technical teams.
- **Enforcement of Architectural Principles:** The AIM visualizes adherence to (or violation of) the core rule of aggregates: external objects must only hold a reference to the Aggregate Root. A dependency arrow pointing to an internal entity of an aggregate is an immediate visual indicator of an architectural violation.
- **Defining Consistency Boundaries:** Aggregates define the boundaries of transactional consistency.[3] Visualizing the interactions between them helps architects reason about distributed transactions, eventual consistency models, and the overall resilience of the system.

In essence, the use of the DDD aggregate as the quantum of analysis transforms the AIM from a simple code dependency graph into a strategic tool for architectural governance. It provides a framework for evaluating coupling, cohesion, and domain boundaries at a level that is directly relevant to the system's business purpose.

1.3 The AIM as a Multi-Layered Dependency Model

A truly comprehensive architectural map cannot be a single, flat representation. Different analysis techniques reveal different facets of a system, and the AIM is designed to integrate

these perspectives into a cohesive, multi-layered model. Each layer provides a unique type of information, and the interplay between these layers yields insights that would be invisible from any single viewpoint. The AIM is composed of three primary layers:

- **Layer 1: Static Dependency Blueprint:** This foundational layer represents the system's *intended* architecture as declared in its static artifacts. It is the blueprint created by developers and architects. This layer is constructed by analyzing artifacts that do not require code execution, such as build manifests (pom.xml, package.json), which declare library dependencies, and API specification files (OpenAPI), which define the formal contracts between services. This layer answers the question: "According to our designs and declarations, how *should* the system be structured?" It represents the potential pathways of interaction and the declared dependencies that form the system's backbone.
- **Layer 2: Dynamic Operational Reality:** This layer represents the system's *actual* behavior as observed while it is running. It is constructed from data collected through dynamic analysis techniques, most notably distributed tracing and application performance monitoring (APM). This layer captures the real-time flow of requests across services, detailing which interfaces are actually called, the frequency of these calls, their latency, and their success or failure rates. This layer answers the question: "In the real world, how *is* the system behaving?" It provides the ground truth of operational dependencies, often revealing "shadow APIs" (undocumented endpoints) or performance bottlenecks that are not apparent from the static blueprint.
- **Layer 3: Logical Consistency Boundaries:** This layer provides the business and domain context that gives meaning to the technical data in the first two layers. It is derived from the application of Domain-Driven Design principles, grouping the services and components identified in the other layers into logical "Aggregates." This layer is often constructed through a combination of automated analysis (e.g., identifying clusters of highly cohesive services) and manual curation by domain experts and architects. It answers the question: "What do these technical components *represent* in terms of business capabilities?" This layer is what transforms the AIM from a technical diagram into a strategic business-architecture tool.

The power of the AIM lies not in any single layer, but in the ability to overlay and compare them. A discrepancy between the Static Blueprint and the Dynamic Reality indicates architectural drift. A cluster of tightly coupled services in the dynamic layer that crosses the logical boundaries defined in the third layer signals a potential flaw in the domain model. By integrating these distinct perspectives, the AIM provides a rich, context-aware view of the system's health, performance, and alignment with business goals.

Section 2: The AST Benchmark: Establishing a Point of

Departure

To fully appreciate the challenge and rationale behind constructing an Aggregated Interface Map without Abstract Syntax Trees (ASTs), it is essential to first establish a benchmark by understanding why ASTs are the conventional and often default choice for deep code analysis. The decision to forgo ASTs is a strategic one, trading the highest possible degree of syntactic fidelity for operational simplicity and language agnosticism. This trade-off shifts the analytical focus from "how the code is precisely written" to the more abstract but cross-platform concerns of "what the code declares and what it ultimately does."

2.1 The Power of ASTs: Why They Are the Default

An Abstract Syntax Tree is a hierarchical data structure that represents the abstract syntactic structure of source code.[11] It serves as the foundational intermediate representation (IR) for a vast array of language processing tools, including compilers, interpreters, linters, and advanced static analysis engines.[12] The process begins with a lexical analyzer (lexer) that scans the raw source code and breaks it down into a sequence of tokens—the smallest meaningful units of a language, such as keywords, identifiers, and operators.[14] A parser then consumes this token stream and, guided by the language's formal grammar, organizes the tokens into a tree structure that reflects the code's logical organization.[16]

The power of the AST lies in its level of abstraction. Unlike a concrete syntax tree (or parse tree), which represents every detail of the source syntax including parentheses, semicolons, and formatting, an AST distills the code down to its essential structural and content-related elements.[11] For instance, a mathematical expression like

(a + b) * c would be represented in an AST as a root node for the multiplication operator (*), with one child being the identifier c and the other child being a sub-tree representing the addition of a and b. The grouping parentheses are implicitly handled by the tree's structure, allowing the analysis tool to focus on the core operations and their relationships rather than superficial syntax.[18] This makes the AST an ideal data structure for performing complex, structure-aware analysis and transformations.

2.2 Information Captured by ASTs (and Lost Without Them)

The richness of the AST data structure allows for the extraction of detailed and highly accurate information about a codebase, much of which is difficult or impossible to obtain reliably through other static means. Forgoing ASTs means accepting a loss of fidelity in these specific areas:

- **Precise Syntactic Structure:** An AST provides an unambiguous representation of the code's grammar. It captures the exact relationships between declarations, statements, expressions, and control flow constructs (like if statements and for loops).[17] This allows an analyzer to understand, for example, that a particular function call occurs inside a conditional block, a piece of context that is lost in simpler representations like a flat list of tokens.
- **Rich Semantic Relationships:** While an AST is initially a purely syntactic representation, it serves as the scaffold for a subsequent phase of analysis known as semantic analysis. During this phase, the tree is traversed, and a symbol table is built to resolve identifiers and determine types.[15] The AST nodes can then be annotated or enriched with this semantic information, linking a variable's usage to its original declaration, determining the precise type of an expression, and understanding variable scope.[20] This semantically-enriched AST is what enables highly accurate dependency tracking, allowing a tool to distinguish between methods with the same name on different classes or to trace the flow of data through a program. Without an AST, achieving this level of semantic understanding is exceptionally difficult.[22]
- **Structure-Aware Transformation and Analysis:** Because an AST models the code's logic, it can be manipulated in powerful ways. Automated refactoring tools can safely rename a variable by traversing the tree and changing only the relevant identifier nodes. Linters and static analysis tools can define rules based on tree patterns to detect code smells or potential bugs, such as finding a resource that is opened but never closed within a specific scope.[12] This ability to reliably traverse and transform the code's logical structure is a key advantage of the AST approach.[19] Without it, analysis is often reduced to less reliable pattern matching on text.

The absence of an AST means that any alternative workflow must find other ways to approximate this information or, more realistically, must operate at a higher level of abstraction where this fine-grained syntactic and semantic detail is not strictly necessary.

2.3 The Rationale for an AST-Free Approach: Complexity and Constraints

Given the power and precision of AST-based analysis, the decision to actively avoid it must be

driven by significant practical challenges. The user's constraint is not arbitrary; it reflects a strategic response to the realities of modern, large-scale software development, particularly in polyglot microservice ecosystems. The primary motivations for an AST-free approach are:

- **Language-Specificity and Maintenance Burden:** The most significant drawback of ASTs is that the parser and the structure of the tree itself are inherently tied to a specific programming language and often a specific version of that language.[24] In a polyglot environment with services written in Java, Python, Go, and TypeScript, an AST-based approach would require developing and maintaining a separate, complex, and robust parser for each language. This is a massive engineering undertaking. Furthermore, as languages evolve (e.g., from Python 3.8 to 3.9), their abstract syntax grammar can change, potentially breaking the analysis tools until they are updated.[25] An AST-free approach seeks a more universal, language-agnostic methodology that can provide a consistent view across the entire ecosystem with a single toolchain.
- **Build Environment Complexity:** To perform the crucial step of semantic analysis (e.g., type resolution across files), an AST-based tool often requires a complete and correctly configured build environment. It needs access to all source files, library dependencies, and compiler flags to accurately resolve symbols and types.[27] Replicating the exact build environment for every service in a large organization for the purpose of analysis can be exceedingly complex and brittle. Failures to resolve dependencies can lead to an incomplete or incorrect AST, rendering the analysis useless.[28]
- **Focus on Architecture over Implementation:** The user's goal is an "Aggregated Interface Map," which implies a focus on the high-level interactions between architectural components, not the line-by-line implementation details within them. For this purpose, the extreme fidelity of an AST may be overkill. The cost of building and maintaining the complex machinery for full AST parsing across an entire organization may not justify the benefits when the primary goal is architectural discovery. Techniques that operate on standardized artifacts like build files, API specifications, or compiled bytecode can provide sufficient architectural insight with a fraction of the implementation complexity.

This strategic choice prioritizes the ability to build a single, unified analysis workflow that can span a diverse system over the ability to perform the deepest possible analysis of any single component. The goal is breadth of coverage and interoperability of the analysis tool itself, accepting a trade-off in the depth of source code analysis.

Section 3: Static Discovery Techniques Beyond the AST

Static analysis involves examining system artifacts without executing them, providing a

blueprint of potential behaviors and declared dependencies. While AST-based analysis offers the highest fidelity, a range of powerful and practical techniques exist that operate on other artifacts. These methods form the foundation of the AIM's static layer, providing a map of the system's intended and declared structure. They can be broadly categorized into four groups: analysis of declarative artifacts, post-compilation analysis, source code surface analysis, and version control history mining.

3.1 Analysis of Declarative Artifacts: The "Ground Truth" of Intent

Perhaps the most reliable and straightforward method for static dependency discovery involves parsing declarative artifacts—files where developers explicitly state their intentions regarding dependencies, interfaces, and configurations. These files are often in standardized, machine-readable formats like XML, JSON, or YAML, making them far easier to parse than general-purpose programming languages.

3.1.1 Build Manifests and Package Managers

The build manifest is the primary source of truth for a project's external and internal library dependencies. By parsing these files, one can construct a foundational dependency graph that shows how services rely on third-party libraries and shared internal modules.

- **Maven (pom.xml):** For Java projects using Maven, the pom.xml file contains a <dependencies> section that explicitly lists each required library's groupId, artifactId, and version. The most effective way to extract the full dependency tree, including transitive dependencies, is to leverage Maven itself using the command mvn dependency:tree.[29] This command resolves the entire graph and can output it in various formats, which can then be parsed to populate the AIM.
- **Gradle (build.gradle):** Gradle files are more complex as they are executable Groovy or Kotlin scripts. While direct parsing is possible, a more robust approach is to use Gradle's maven-publish plugin to generate a pom.xml file, which can then be analyzed as above.[30] Alternatively, custom Gradle tasks can be written to inspect the configurations object and programmatically extract dependency information.[31]
- **NPM (package.json):** For Node.js projects, the package.json file contains dependencies and devDependencies objects that list required packages and their semantic version ranges.[34] Parsing this JSON file is trivial, and tools like npm audit can be used to analyze the full dependency tree for vulnerabilities and other metadata.[36] It is crucial to distinguish between production dependencies, which are part

of the running application, and development dependencies, which are only used for testing and building.[37]

3.1.2 API Specification Files

For a map of *interfaces*, API specification files are the most critical declarative artifact. They define the formal contract of a service.

- **OpenAPI (Swagger):** The OpenAPI Specification (OAS) has become the de facto standard for describing REST APIs.[39] These specifications, written in YAML or JSON, detail every available endpoint, the supported HTTP methods, expected parameters, request and response schemas, and authentication methods.[40] Parsing these files provides a precise, high-fidelity map of a service's public-facing interface. Numerous libraries, such as Swagger Parser for Java, exist to programmatically consume these specifications and extract their components into structured objects, making it straightforward to identify all defined operations and data models.[42] This method discovers the *intended* and *documented* interface, which serves as a crucial baseline for comparison against dynamically observed behavior.

3.1.3 Build and Deployment Configurations

Modern software systems are defined not just by their code but by their build and deployment configurations. These files often contain implicit dependency information.

- **CI/CD Pipelines:** Files like Jenkinsfile, GitHub Actions workflows (.github/workflows/*.yml), or GitLab CI (.gitlab-ci.yml) define the build, test, and deployment process. Analyzing these files can reveal the order in which services are built, dependencies between build jobs, and the environments to which services are deployed.
- **Infrastructure-as-Code (IaC):** Files like docker-compose.yml or Kubernetes deployment manifests explicitly define how services are networked together. They specify service names, container images, port mappings, and often network policies that dictate which services are allowed to communicate with each other. This provides a rich source of information about the intended runtime topology of the system. Tools that wrap the build process, such as scan-build, exemplify the principle of extracting information from the build itself, which can be extended to these configuration files.[45]

3.2 Post-Compilation Analysis: Examining the Executable Blueprint

Analyzing compiled artifacts offers a powerful, language-agnostic approach that moves one step closer to the runtime reality of the system. Instead of parsing the varied and complex syntaxes of high-level languages, this method analyzes the standardized intermediate representation (IR) or bytecode that the compiler produces.

- **JVM Bytecode (.class files):** For any language running on the Java Virtual Machine (Java, Kotlin, Scala, etc.), the compiler produces standardized .class files containing Java bytecode. This bytecode is a set of instructions for the JVM and contains a wealth of structural information, including class definitions, method signatures, field declarations, and, most importantly, direct instructions for method invocations (invokevirtual, invokestatic, etc.).[47] By analyzing this bytecode, it is possible to construct a highly accurate call graph showing which methods call which other methods, both within the same service and across library boundaries. Tools like the ASM library, SpotBugs (the successor to FindBugs), and the Python-based
  coffea library are designed specifically for this purpose, enabling deep dependency analysis without ever looking at the original source code.[49] This method is particularly valuable because it reflects what the compiler
  actually produced, accounting for compiler-generated code or optimizations that might not be obvious from the source.
- **Python Bytecode (.pyc files):** Python also uses a compilation step, translating .py source files into .pyc files containing bytecode for the Python virtual machine. These files are often created automatically by the interpreter. The import statements in the source code are translated into IMPORT_NAME and IMPORT_FROM opcodes in the bytecode. By disassembling .pyc files, one can reliably determine which modules are imported by a given file.[53] This is more robust than text-based scanning of
  .py files, as it is not fooled by imports inside comments or multiline strings.

The primary advantage of post-compilation analysis is its combination of high fidelity and language independence (at the IR level). It provides a view of the code that has been normalized by the compiler, abstracting away the syntactic sugar of the source language.

3.3 Source Code Surface Analysis: High-Speed, Lower-Fidelity Methods

This category includes techniques that analyze the source code text directly but deliberately avoid the complexity of building a full, grammatically correct AST. These methods trade some accuracy for significant gains in speed and implementation simplicity.

3.3.1 Token-Stream Analysis

A significant step up from simple text matching is token-stream analysis. This approach uses only the first phase of a traditional compiler—the lexical analyzer (or lexer)—to convert the source code into a linear sequence of tokens.[54] A tool like ANTLR can be configured to generate a lexer for a given language, which will reliably identify keywords (

import, class), identifiers (MyClass), operators, and literals, while correctly ignoring comments and whitespace.[55]

Once the code is represented as a token stream, simple algorithms can be used to scan for dependency-related patterns. For example, one can search for the sequence of tokens `` to find import statements. This is far more robust than regex because the lexer understands the basic building blocks of the language. It will not, for instance, mistakenly identify an "import" statement inside a string literal. This technique provides a good balance of speed and accuracy for discovering high-level dependencies like module imports or function definitions, without the overhead of full parsing.[57] ANTLR's ability to define channels for tokens (e.g., to send comments and whitespace to a hidden channel) makes it easy to focus the analysis only on the code-relevant tokens.[58]

3.3.2 Regular Expression (Regex) Based Parsing

Using regular expressions to parse source code is the fastest and simplest technique to implement, but it is also by far the most brittle and error-prone. It should be approached with extreme caution and is generally not recommended for building a reliable AIM.

A simple regex can be used to find lines containing keywords like import or require.[60] However, this approach is fundamentally flawed because programming languages are typically context-free languages, whereas regular expressions can only correctly parse regular languages.[62] This theoretical limitation has profound practical consequences:

- **Inability to Handle Nesting:** Regex cannot handle arbitrarily nested structures, such as nested code blocks or parenthesized expressions.[64]

- **Context Blindness:** A regex has no understanding of context. It cannot distinguish between an import keyword in executable code and the same word appearing inside a comment or a string literal, leading to a high rate of false positives.[65]
- **Complexity and Unmaintainability:** Creating a regex that attempts to account for all edge cases (e.g., multiline statements, escaped characters in strings) results in an incredibly complex and unreadable expression that is nearly impossible to maintain or debug.[67]

While regex can be a useful tool for ad-hoc searching or simple log parsing [61], relying on it to build an accurate architectural map is a significant risk. The resulting data is likely to contain both false positives (detecting dependencies that don't exist) and false negatives (missing dependencies that do), undermining the credibility and utility of the entire AIM.[65]

3.4 Version Control History Mining: Analyzing API Evolution

A novel form of static analysis involves mining the history stored in the version control system, typically Git. The commit history is a rich dataset that documents the evolution of the codebase, including changes to dependencies and interfaces.

By programmatically analyzing the history of key files, one can add a temporal dimension to the AIM. For example, running git log --follow -p -- <path-to-pom.xml> will show every change made to the project's dependencies, including when they were added, removed, or updated.[71] Similarly, tracking the history of an OpenAPI specification file reveals the entire lifecycle of an API: when endpoints were introduced, when parameters were changed (potentially indicating a breaking change), and when versions were deprecated.[72]

This analysis provides valuable context that is unavailable from a single snapshot of the code. It helps answer questions like:

- How stable is this service's interface?
- Which dependencies are legacy and which have been recently introduced?
- Can we correlate a recent performance degradation with a specific library version update?

Tools and APIs provided by platforms like GitHub can further automate this process, allowing for the retrieval of dependency diffs between commits.[74] This historical perspective is crucial for understanding architectural trends, identifying areas of high churn, and managing technical debt over time.[75]

The various static methods form a "hierarchy of truth" that a sophisticated AIM workflow must

reconcile. Declarative artifacts like pom.xml or OpenAPI specs represent the developer's *intent*. Source code surface analysis offers a glimpse into the *implementation*. Post-compilation bytecode analysis reveals the *compilation reality*. These layers can, and often do, conflict. A developer might forget to add a library to the build manifest even though it is imported in the code. An API endpoint might be implemented and callable but missing from the official OpenAPI documentation. A compiler might optimize away a function call that is visible in the source. An effective AIM must not only choose one source of truth but must ingest data from all of them and use their discrepancies as a source of powerful insights. These conflicts are not errors in the analysis; they are valuable signals of documentation drift, unused dependencies, or "shadow" infrastructure that require architectural attention.

Section 4: Dynamic Discovery: Mapping the System in Motion

While static analysis provides a crucial blueprint of a system's potential dependencies, it cannot capture the reality of a system in operation. Dynamic discovery techniques address this by observing the software as it runs, mapping the actual communication paths, and measuring their performance characteristics. This approach is essential for validating the static blueprint, uncovering emergent behaviors, and enriching the Aggregated Interface Map with real-world operational data. These techniques can be broadly divided into two categories: active instrumentation, which modifies the application to report on its own behavior, and passive observation, which monitors the system from the outside.

4.1 Active Instrumentation and Tracing: The Gold Standard for Observability

Active instrumentation involves using agents or libraries to inject monitoring code directly into a running application. This "inside-out" view provides the highest fidelity data on application behavior, forming the gold standard for dynamic analysis.

4.1.1 Distributed Tracing and Application Performance Monitoring (APM)

In a microservices architecture, a single user request can trigger a complex cascade of calls across dozens of services. Distributed tracing is a technique designed to track the entire journey of such a request.[77] The core concepts are:

- **Trace:** Represents the end-to-end journey of a single request, identified by a unique TraceId.
- **Span:** Represents a single unit of work or operation within a trace (e.g., an HTTP call, a database query), identified by a SpanId. Spans are organized into a parent-child hierarchy, forming the structure of the trace.[78]

Industry standards like **OpenTelemetry** provide a vendor-neutral set of APIs, SDKs, and agents for collecting trace data.[78] APM platforms such as Datadog, New Relic, and Dynatrace leverage these standards to provide powerful observability solutions.[80] They use

auto-instrumentation, where an agent automatically detects popular frameworks (like Spring Boot in Java or Express in Node.js) and libraries (like JDBC for databases or Kafka for messaging) and injects the necessary code to start, propagate, and end spans without requiring manual code changes.[82]

For the AIM, distributed tracing data is invaluable. It directly reveals:

- **Service-to-Service Dependencies:** Every remote call between services is captured as a span, definitively mapping the runtime dependency graph.
- **External Dependencies:** Calls to databases, message queues, and third-party APIs are also captured, providing a complete picture of the service's ecosystem.
- **Performance Metadata:** Each span is enriched with critical performance data, including latency (duration), status (success/error), and other attributes. This allows the AIM to not just show a connection between two services, but to characterize that connection (e.g., "Service A calls Service B's /users endpoint 100 times per minute with an average latency of 50ms and a 1% error rate").[83]

The primary challenge with this approach is the potential for performance overhead and the sheer volume of data generated, which is often managed through sampling strategies.[85]

4.1.2 Dynamic Binary Instrumentation (DBI)

For scenarios requiring even deeper or more targeted analysis, particularly for legacy applications, third-party software, or security-hardened binaries where source code is unavailable, Dynamic Binary Instrumentation (DBI) toolkits are employed.

- **Frida:** Frida is a world-class DBI toolkit that allows an analyst to inject JavaScript snippets into running processes on a wide range of operating systems.[86] Using Frida, one

can
hook arbitrary functions within a process, intercepting their arguments and return values, inspecting and modifying memory, and tracing private, non-exported API calls.[87] This is exceptionally powerful for reverse engineering communication protocols or understanding the internal workings of a black-box component. However, it is a highly specialized and intrusive technique, often reserved for deep-dive security analysis or debugging rather than broad architectural discovery due to its complexity and performance impact.[87]

- **Java Agents:** The Java Virtual Machine (JVM) provides a standard mechanism called the **Java Instrumentation API**. A Java Agent is a special JAR file that can be attached to a running JVM (either at startup or dynamically) to transform the bytecode of classes as they are loaded.[89] This is the core technology that powers most Java APM agents. It allows for the modification of method bodies to add timing code, capture parameters, and create trace spans without altering the original source code.[90]

4.2 Passive Observation: Inferring Behavior Without Intrusion

Passive observation techniques aim to discover system topology and behavior by monitoring the environment from the outside, avoiding the overhead and potential instability of injecting code into the application itself.

4.2.1 Network Traffic Analysis

This method involves capturing and analyzing the network packets that flow between services. By inspecting the source and destination IP addresses and ports, it is possible to build a map of which services are communicating with each other.

- **Service Mesh:** Modern cloud-native platforms often use a service mesh like Istio or Linkerd. The service mesh deploys a "sidecar" proxy alongside each service instance. All inbound and outbound network traffic for the service is routed through this proxy. The proxy can then export detailed telemetry about the traffic, including source, destination, protocol, latency, and success/error rates, providing a rich source of dependency information without touching the application code.[91]
- **eBPF (extended Berkeley Packet Filter):** A revolutionary technology within the Linux kernel, eBPF allows for safe, sandboxed programs to be attached to various kernel hooks, including those related to networking.[92] eBPF-based monitoring tools can efficiently capture network traffic at the kernel level, correlating packets to specific processes. This

provides a highly performant way to map network communications across an entire host or cluster with minimal overhead.[93]

The primary limitation of network traffic analysis is its potential blindness to application-layer context, especially when traffic is encrypted with TLS. While it can show that Service A is talking to Service B, it may not be able to reveal the specific REST endpoint or gRPC method being called without more advanced (and intrusive) techniques like TLS interception.

4.2.2 Service Discovery Registries

In dynamic microservice environments, where instances are constantly being created and destroyed, services rely on a **Service Registry** to find each other. When a service instance starts up, it registers its network location (IP address and port) and other metadata with a central registry like Consul, Eureka, or Zookeeper.[95] Other services (consumers) can then query this registry to discover the current location of the services they need to call.[98]

By querying the API of the service registry, the AIM system can obtain a real-time, comprehensive inventory of all active service instances in the environment.[99] This doesn't directly map the

interactions between services, but it provides a live "census" of all the potential nodes in the dependency graph, which is a critical piece of the puzzle.

A fundamental trade-off exists across these dynamic techniques, balancing the depth of observability against the performance overhead and operational risk. DBI with Frida offers unparalleled depth but is highly intrusive and complex, making it suitable for targeted analysis in development environments. APM-style auto-instrumentation provides rich, application-level context with moderate overhead, making it the standard for staging and often used with sampling in production. Passive network monitoring offers the lowest overhead and risk, providing a high-level topological view, but often lacks application-specific detail. A mature AIM strategy does not choose one of these but orchestrates a portfolio. It might use passive monitoring for a low-impact, always-on view of production topology, leverage APM with adaptive sampling for performance-aware tracing, and reserve DBI for deep-dive forensic investigations of specific, problematic services in a controlled, non-production setting. The AIM must be capable of ingesting and correlating data from all these sources to build a truly comprehensive picture of the system in motion.

Section 5: A Comparative Framework for Analysis

Techniques

The creation of a comprehensive Aggregated Interface Map requires a multi-faceted approach, leveraging a portfolio of analysis techniques. No single method is sufficient to capture the full complexity of a modern software system. To guide the strategic selection and combination of these techniques, this section presents a comparative framework that evaluates each non-AST method against a set of critical criteria. This analysis illuminates the inherent trade-offs between fidelity, performance, complexity, and coverage, enabling architects to design a tailored AIM workflow that aligns with their specific goals, resources, and risk tolerance.

5.1 Establishing Evaluation Criteria

Before comparing the techniques, it is essential to define the dimensions of comparison. The following criteria provide a structured basis for evaluating the suitability of each method for dependency discovery:

- **Accuracy & Fidelity:** This measures the precision and reliability of the dependency information generated. It considers the likelihood of false positives (identifying dependencies that do not exist) and false negatives (missing dependencies that do exist). High-fidelity techniques provide detailed, context-rich information (e.g., specific method calls), while low-fidelity techniques may only provide high-level connections (e.g., service-to-service communication).
- **Performance Overhead:** This assesses the impact of the analysis on system resources. For static techniques, this refers to the impact on build times and the computational cost of the analysis itself. For dynamic techniques, it refers to the runtime impact on application performance (CPU, memory, and network latency).
- **Implementation Complexity:** This evaluates the level of effort and expertise required to set up, configure, and maintain the analysis tool or process. This includes factors like the need for specialized knowledge, integration with build systems or runtime environments, and ongoing maintenance.
- **Language Independence:** This measures the technique's applicability across a polyglot codebase. A highly language-independent technique can be applied uniformly to services written in different programming languages, which is a key requirement for a unified AIM workflow.
- **Coverage & Blind Spots:** This identifies the inherent limitations of each technique—the types of dependencies or system behaviors that it is incapable of detecting.[101] Understanding these blind spots is crucial for combining techniques to achieve

comprehensive coverage.[103]

- **Primary Use Case:** This defines the scenario or goal for which the technique is most effective and appropriate, guiding its placement within a broader analysis strategy.

5.2 The Comparative Analysis Matrix

The following table synthesizes the analysis of the non-AST techniques discussed in Sections 3 and 4, evaluating them against the established criteria. This matrix serves as a central reference for understanding the strengths and weaknesses of each approach.

Table 5.1: Comparative Analysis of Non-AST Dependency Discovery Techniques

Technique	Type	Accuracy & Fidelity	Performance Overhead	Implementation Complexity	Language Independence	Coverage & Blind Spots	Primary Use Case
Build Manifest Parsing	Static	High (for declared libs)	Low (fast build step)	Low (uses build tools)	High (manifest format is standard)	Misses undeclared dependencies, code-level calls, and usage context.	Foundational library dependency mapping and Software Bill of Materials (SBOM) generation.
API Spec Parsing	Static	Very High (for intended	Negligible	Low (uses standard parsers	Very High (OAS is a standar	Misses undocumented ("shadow")	Defining and validating service

		contract)		)	d)	APIs, implementation bugs, and actual usage patterns.	contracts and intended public interfaces.
Bytecode Analysis	Static	High (reflects compiled reality)	Medium (slower than text scan)	High (requires deep VM knowledge)	High (targets standard bytecode)	Misses dynamically loaded code, reflection-based calls, and configuration-driven behavior.	Accurate, deep code-level dependency discovery in compiled languages without source.
Token-Stream Analysis	Static	Medium (better than regex)	Low (faster than parsing)	Medium (requires lexer setup)	Medium (requires per-language lexer)	Misses semantic context, type information, and polymorphism. Can be fooled by complex	Fast, "good enough" source scanning for imports and high-level call patterns.

						macros or syntax.	
Regex-Based Search	Static	Very Low (brittle, high error rate)	Very Low (very fast)	Low (deceptively simple)	High (text is universal)	Fails on nested structures, comments, strings, and context. Extremely unreliable for parsing.	Ad-hoc keyword searching; not recommended for systematic dependency mapping.
Distributed Tracing (APM)	Dynamic	Very High (for executed paths)	Medium-High (agent overhead, sampling needed)	Medium (requires agent deployment/config)	High (agents for many languages)	Misses unexecuted code paths ("dead code") and dependencies in non-instrumented code.[104]	Understanding real-world service interactions, performance bottlenecks, and user flows.
DBI (e.g., Frida)	Dynamic	Highest (function/memory level)	Very High (intrusive, can alter behavio	Very High (requires expert knowledge)	High (operates on binaries/processes)	Can be detected/blocked by security measur	Deep reverse engineering, security analysis

			r)			es. Complex setup. Limited scalability.	, and debugging of black-box components.
Network Traffic Analysis	Dynamic	Low-Medium (topology, not payload)	Low (passive monitoring)	High (requires network taps/service mesh)	Very High (protocol-based)	Blind to encrypted traffic payload, intra-process calls, and application-level logic.	Discovering high-level service topology and communication patterns with minimal intrusion.
Service Registry Querying	Dynamic	High (for service inventory)	Very Low (simple API call)	Low	Very High (registry API is standard)	Only shows available services, not their interactions or dependencies.	Real-time service discovery and inventory management for dynamic environments.

5.3 In-Depth Discussion of Trade-offs

The analysis matrix reveals several fundamental trade-offs that must be managed when designing an AIM workflow. There is no single "best" technique; the optimal approach is a carefully orchestrated combination of methods chosen to complement each other's strengths and mitigate their weaknesses.

5.3.1 Static vs. Dynamic Analysis: Blueprint vs. Reality

The most significant trade-off is between static and dynamic analysis. Static analysis examines the system's artifacts at rest, creating a map of its *potential* state and all possible execution paths.[104] It is comprehensive in that it can analyze every line of code, including error-handling paths that are rarely executed. Its primary strength is in finding vulnerabilities and design flaws within the code itself, before the application is even run.[105] However, it is blind to the runtime environment, configuration, and actual usage patterns.

Dynamic analysis, conversely, observes the system in motion, mapping its *actual* behavior under a specific workload.[104] It excels at finding runtime issues like performance bottlenecks, memory leaks, and vulnerabilities that only emerge from the interaction of components in a live environment.[107] Its fundamental limitation is that it can only observe the code paths that are actually executed during the test; it has a blind spot for "dead" or untested code.[104] A comprehensive security and architectural strategy requires both: static analysis to secure the blueprint and dynamic analysis to secure the running implementation.[105]

5.3.2 Accuracy vs. Effort Spectrum

The techniques exist on a spectrum of accuracy versus implementation effort. At one end, regex-based searching is trivial to implement but yields highly inaccurate and unreliable results. At the other end, Dynamic Binary Instrumentation with Frida provides the highest possible fidelity, down to the memory level, but requires immense expertise and is complex to deploy at scale. In between, techniques like build manifest parsing and API specification parsing offer a "sweet spot" of very high accuracy for their specific domains with relatively low implementation complexity. A pragmatic approach to building an AIM would start with these high-ROI techniques and progressively layer on more complex methods like bytecode analysis and distributed tracing as the need for greater detail arises.

5.3.3 The Polyglot Challenge and Language Independence

For organizations with a heterogeneous technology stack, the "Language Independence" criterion is paramount. Techniques that operate on standardized, cross-language artifacts are the most valuable for creating a unified AIM.

- **High Independence:** API Specification Parsing (OAS is a language-agnostic standard), Network Traffic Analysis (operates on network protocols), and Service Registry Querying (operates on standard registry APIs) are almost completely independent of the programming languages used by the services.
- **Medium Independence:** Bytecode Analysis is highly effective for families of languages that compile to a common intermediate representation (e.g., all JVM languages).
- **Low Independence:** Token-Stream Analysis requires a language-specific lexer for each language in the ecosystem, increasing the maintenance burden.

This analysis reveals that a truly robust AIM strategy cannot rely on a single tool or technique. It must be a data fusion platform. The blind spots of one method are often the strengths of another. For instance, network analysis can identify that two services are communicating, while distributed tracing can reveal the specific API call being made over that connection. Build manifest analysis can declare a dependency on a library, and bytecode analysis can confirm which of its functions are actually called from the code. The most powerful insights arise from combining and comparing the data from this portfolio of tools, using each to validate and enrich the others. This leads to a model where the AIM is not just a map, but an analytical engine capable of answering complex queries that span both the static and dynamic views of the system.

Section 6: The Hybrid AIM Workflow: A Strategic Synthesis

Having analyzed the strengths, weaknesses, and trade-offs of various non-AST analysis techniques, this section synthesizes these findings into a practical, phased workflow for constructing and maintaining a living Aggregated Interface Map. This hybrid approach is designed to be implemented incrementally, delivering value at each stage while building towards a comprehensive, automated system for architectural discovery and governance. The workflow transforms the AIM from a static, one-time diagram into a dynamic, data-driven model that reflects the true state of the software system and evolves with it. The most

powerful capability of this workflow is not merely mapping dependencies but actively detecting "architectural drift"—the divergence between the intended design and the operational reality.

6.1 A Phased Approach to AIM Construction

The construction of the AIM is best approached in three distinct phases, each building upon the last. This allows an organization to start with low-effort, high-impact techniques and progressively add more sophisticated layers of analysis.

6.1.1 Phase 1: Foundational Static Mapping (The Blueprint)

This initial phase focuses on gathering data from declarative and compiled artifacts to create a baseline map of the system's intended and declared structure. This "blueprint" serves as the foundational layer of the AIM.

1. **Automated Dependency Ingestion:** The process begins by setting up automated scanners that traverse the organization's source code repositories. These scanners identify and parse build manifest files. For Maven projects, they execute mvn dependency:tree to resolve the complete dependency graph, including transitive dependencies.[29] For NPM projects, they parse package.json files to extract dependencies and devDependencies.[36] The extracted data, which forms a graph of library-to-service and library-to-library dependencies, is ingested into a central data store, preferably a graph database.
2. **API Contract Ingestion:** In parallel, the scanners locate and parse all API specification files, primarily OpenAPI/Swagger documents.[111] Using standard parsers, the system extracts every defined endpoint, operation, data schema, and security requirement.[42] This information is used to construct a detailed map of the system's formal, documented interfaces. Each service node in the graph is annotated with the interfaces it provides and consumes.
3. **Bytecode and Binary Scanning:** For services written in compiled languages (e.g., Java/JVM, Python), the CI process is augmented to scan the build artifacts (.jar, .war, .pyc files). Bytecode analysis tools are used to discover class-level dependencies, method calls, and inheritance hierarchies.[51] This data is then cross-referenced with the information from the build manifests. This step validates that the declared dependencies are actually used in the compiled code and can uncover dependencies that were not

explicitly declared.

At the end of Phase 1, the organization will have a rich, static map of its software ecosystem. This map shows all declared library dependencies and all documented API contracts. Even at this stage, it provides immense value for tasks like security vulnerability scanning (by identifying all services using a vulnerable library) and impact analysis (by showing which services consume a given API).

6.1.2 Phase 2: Dynamic Validation and Enrichment (The Reality)

This phase brings the static blueprint to life by overlaying it with data from the running system. This is where the map transitions from representing potential connections to representing actual, operational interactions.

1. **Deploy Instrumented Services:** The key prerequisite for this phase is the widespread deployment of APM agents, ideally based on the OpenTelemetry standard for vendor neutrality. These agents should be enabled by default in all pre-production (e.g., staging, QA) environments and, where performance allows, in production (often with sampling enabled).[80] Auto-instrumentation capabilities will handle the bulk of the work for common frameworks and protocols.
2. **Capture and Aggregate Traces:** As applications are used—either through automated integration tests, QA activities, or real user traffic—the APM agents capture distributed traces for the requests flowing through the system. This trace data, which details the exact sequence of service calls, database queries, and other interactions for each request, is sent to an observability backend for aggregation and analysis.
3. **Data Correlation and Enrichment:** The aggregated dynamic data is then correlated with the static map created in Phase 1. This crucial step enriches the AIM in several ways:
    - **Validation:** It confirms which of the potential dependencies and API calls identified in the static map are actually being used in practice. Edges in the graph that are present in the static map but never appear in dynamic traces (over a significant period) may represent dead code or unused dependencies.
    - **Discovery:** It uncovers dependencies that were *not* present in the static map. These are often "shadow" API calls—undocumented, informal, or private APIs being used between services. Identifying these is a major win for architectural governance.
    - **Performance Enrichment:** Each edge in the dependency graph is annotated with key performance indicators (KPIs) derived from the trace data, such as call frequency (throughput), average/p95 latency, and error rate.[83] The static connection "Service A can call Service B" is transformed into the dynamic insight "Service A calls Service B's
      checkout endpoint 50 times/sec with a p95 latency of 300ms and a 0.5% error rate."

6.1.3 Phase 3: Continuous Integration and Evolution

The final phase transforms the AIM from a project into a continuous process, embedding it into the daily engineering workflow to ensure it remains accurate and provides ongoing value.

1. **Automate Analysis in CI/CD:** The static analysis steps from Phase 1 are integrated directly into the Continuous Integration (CI) pipeline.[112] When a developer pushes a change to a
   pom.xml, package.json, or an OpenAPI specification, a CI job is automatically triggered. This job re-scans the changed artifact, updates the static layer of the AIM in the central graph database, and can even perform validation checks. For example, the build could fail if a change introduces a dependency that violates a predefined architectural rule (e.g., a service in the "presentation" layer attempting to depend directly on a service in the "data" layer of another domain).
2. **Continuous Dynamic Monitoring:** The dynamic data collection from Phase 2 is not a one-time event. It is an always-on process, continuously streaming telemetry from running environments into the observability platform and, from there, into the AIM's data store. This ensures the map always reflects the current operational state of the system.
3. **Automated Drift Detection and Alerting:** The most advanced capability of the workflow is automated drift detection. The system continuously compares the static blueprint with the dynamic reality. When a significant discrepancy is found, an alert is generated. Examples of drift alerts include:
   - **"New Shadow API Detected":** A new dependency path appears in the dynamic traces that does not correspond to any documented API in the static map.
   - **"Deprecated API In Use":** Dynamic traces show calls to an API endpoint that has been marked as deprecated in its OpenAPI specification.
   - **"Potential Dead Dependency":** A library is declared in a build manifest but no calls to its code are observed in dynamic traces over an extended period (e.g., 30 days).

This hybrid workflow creates a powerful feedback loop for architects and developers. It moves architectural governance from a process of manual reviews and documentation to an automated, data-driven system that provides real-time insights into the health, performance, and integrity of the software architecture.

Section 7: Visualizing the Map: From Data to Architectural Insight

The data collected through the hybrid AIM workflow—a rich, multi-layered graph of static declarations, dynamic interactions, and performance metrics—is immensely valuable, but its raw form is often too complex for direct human consumption. The final, critical step in making the Aggregated Interface Map a truly effective tool is visualization. An effective visualization is not merely a static picture of the system; it is an interactive analysis interface designed to make complex architectural patterns, problems, and insights visually intuitive to a human observer.[114] Moving beyond simple node-link diagrams is essential to avoid the "hairball" effect common with large graphs and to convey the multiple dimensions of the AIM data.

7.1 Beyond the Node-Link Diagram: The Need for Advanced Metaphors

For any non-trivial system, a simple, force-directed layout of all services and their dependencies will quickly devolve into an incomprehensible tangle of nodes and edges, often referred to as a "hairball".[115] To overcome this, the AIM visualization should employ more sophisticated metaphors that leverage human intuition to communicate structure and meaning. A metaphor in visualization maps the characteristics of a well-understood source domain (like a city or a map) to the target domain of software architecture, making abstract relationships tangible.[116]

7.2 Applying Visualization Metaphors to the AIM

Several powerful metaphors can be applied to visualize the different layers and dimensions of the AIM data, transforming it from a raw graph into an insightful architectural landscape.

7.2.1 Layered Graph / Levelized Structure Maps

One of the most effective techniques for visualizing dependencies is to organize the components into hierarchical layers. This approach, used in tools like Structure101, arranges components such that dependencies flow primarily in one direction (e.g., downwards).[117] For the AIM, this would involve assigning each aggregate or service to a predefined architectural

layer (e.g., User Interface, API Gateway, Business Services, Data Persistence).

The visualization would then render these layers explicitly. The vast majority of dependency edges should point from a higher layer to a lower one. Any edge that points upwards (a "reverse" dependency) or skips multiple layers immediately stands out as a potential architectural violation, such as a business service improperly calling a UI component. This layout makes architectural layering rules visually explicit and deviations immediately obvious.

7.2.2 The "Software City" Metaphor

The Software City metaphor provides a rich, multi-dimensional view of the system by mapping architectural concepts to urban elements.[117]

- **Districts:** Logical domains or DDD Aggregates can be represented as distinct districts within the city.
- **Buildings:** Individual microservices or components are visualized as buildings within their respective districts. The dimensions of a building can be mapped to various metrics: its height could represent code complexity (e.g., lines of code), its footprint could represent memory usage, and its color could indicate its health status (green, yellow, red).
- **Roads:** The dependencies and API calls between services are represented as roads connecting the buildings. The thickness or traffic flow on these roads can be directly mapped to the throughput data collected from dynamic tracing, showing which communication paths are the most heavily used.

This metaphor is highly intuitive. A district with a single, towering skyscraper that all roads lead to is instantly recognizable as a potential monolith or a critical single point of failure. A district with crumbling, red-colored buildings indicates a problematic area of the system.

7.2.3 Cartographic and Biological Metaphors

For even more abstract or exploratory visualizations, other metaphors can be drawn from cartography and biology.

- **Cartographic Maps:** The system can be visualized as a geographic map, where services are cities, and the underlying infrastructure (e.g., cloud regions, Kubernetes clusters) forms the terrain or continents.[118] The "trade routes" (API calls) between cities can be colored or weighted by data volume or latency. This metaphor is particularly effective for visualizing geographically distributed systems.

- **Biological Systems:** A software ecosystem can be viewed as a biological one, with services as organisms and dependencies as the food web. Alternatively, it can be modeled as a neural network, emphasizing the complex, emergent patterns of interaction rather than a rigid, planned structure.[120] These metaphors are best suited for analyzing the resilience and dynamic behavior of highly complex, self-organizing systems.

7.3 Key Features of an Interactive AIM Visualization Tool

Regardless of the chosen metaphor, the visualization must be interactive to serve as an effective analysis tool. A static image is insufficient. The following features are critical:

- **Filtering and Searching:** The user must be able to slice and dice the view. For example: "Show only the dependencies for the 'Payments' aggregate," "Highlight all services that use a vulnerable version of log4j," or "Filter for API calls with a P95 latency greater than 500ms."
- **Zooming and Hierarchical Exploration:** The tool should support seamless zooming from a high-level, system-wide landscape view down to the level of a single aggregate, and then further into the specific API endpoints and data schemas of an individual service.[122]
- **Data Overlay Toggling:** Users should be able to toggle different layers of information on and off. One might start with the static dependency view, then overlay the dynamic traffic data to see which connections are active, and finally add a performance "heatmap" overlay to identify latency hotspots.
- **Temporal View (Time Slider):** By incorporating the data from version control history mining, the tool can include a time slider. This would allow an architect to scrub back and forth through time, visualizing how the system's architecture has evolved, how new dependencies were introduced, and how components were refactored over months or years.

7.4 Recommended Tools and Libraries

Building a sophisticated, interactive AIM visualization requires a combination of a robust backend for data storage and powerful frontend libraries for rendering.

- **Data Storage:** The complex, interconnected nature of the AIM data makes a **graph database** (such as Neo4j, TigerGraph, or Amazon Neptune) the ideal choice for the backend. A graph database naturally models the nodes (services, aggregates, libraries) and edges (dependencies, API calls) of the system, allowing for efficient querying of

complex relationships (e.g., "find all transitive downstream consumers of this deprecated API").

- **Visualization Engine:**
  - **High-Level Tools:** For rapid prototyping and analysis, tools like **Gephi** (an open-source platform for graph visualization) [123] or Python libraries like **NetworkX** [124] can be used to generate initial visualizations.
    Structurizr is an excellent tool specifically designed for the C4 model, which aligns well with the layered, hierarchical nature of the AIM.[122]
  - **Low-Level Libraries:** For a fully custom, bespoke, and highly interactive web-based visualization, **D3.js** is the industry standard.[125] While it has a steeper learning curve, it provides unparalleled flexibility to implement any of the advanced metaphors and interactive features described above, creating a truly powerful and tailored analysis tool.

By investing in a rich, interactive visualization, an organization transforms the AIM from a dataset into a decision-making platform, enabling architects and developers to intuitively understand, question, and improve their software systems.

Section 8: Conclusion and Strategic Recommendations

8.1 Summary of Findings: The Viability of a Non-AST Approach

This report has systematically deconstructed the concept of an Aggregated Interface Map (AIM) and explored a comprehensive array of techniques for its construction, under the key constraint of avoiding Abstract Syntax Trees. The analysis concludes that an AST-free approach is not only viable but, for the purposes of architectural discovery in modern, polyglot distributed systems, is arguably superior to traditional, single-language static analysis methods.

The constraint of avoiding ASTs forces a strategic shift away from deep, language-specific syntactic analysis towards a more holistic, system-level perspective. This leads to a hybrid methodology that is inherently more resilient to technological diversity and better aligned with the challenges of microservice architectures. The core strength of the proposed workflow lies in its fusion of two distinct but complementary sources of truth: the **static blueprint** (the

system's intended design as captured in declarative artifacts and compiled code) and the **dynamic reality** (the system's actual behavior as observed in a running environment). This combination yields a richer, more accurate, and more context-aware model of the system than either approach could provide in isolation. The ability to automatically detect and highlight "architectural drift" between these two views transforms the AIM from a passive documentation tool into an active architectural governance engine.

8.2 Strategic Recommendations for Implementation

To successfully implement an Aggregated Interface Map within an organization, a phased and strategic approach is recommended. This allows for incremental value delivery and builds momentum for the initiative.

1. **Start with Declarative Artifacts:** The journey should begin with the "low-hanging fruit." Implement automated scanning and parsing of build manifests (pom.xml, package.json, etc.) and API specification files (OpenAPI/Swagger). This initial step is relatively low in complexity but provides immediate, high-value insights into library dependencies and documented service contracts. This foundational static map can be used straight away for dependency vulnerability scanning and basic impact analysis.
2. **Layer on Dynamic Tracing:** The next logical step is to introduce dynamic analysis, starting in a controlled pre-production environment. Deploy APM agents using a standardized framework like OpenTelemetry across services. The collected trace data will validate the static map, uncover "shadow" APIs, and enrich every dependency link with critical performance metrics (latency, throughput, error rates). This phase provides the "ground truth" that contextualizes the static blueprint.
3. **Invest in a Unified Data Model:** It is critical to avoid treating the outputs of the various analysis tools as separate, siloed reports. A central **graph database** should be established as the core of the AIM system. This unified data model will ingest and fuse the data from all static and dynamic sources, enabling powerful, cross-domain queries that are impossible when data is fragmented. This investment is the key to unlocking the most profound architectural insights.
4. **Prioritize Automation in CI/CD:** To prevent the AIM from becoming an outdated artifact, its maintenance must be automated. Integrate the static analysis and drift detection processes directly into the CI/CD pipeline. Changes to dependencies or API contracts should trigger automatic updates to the map and run against a set of predefined architectural rules. This embeds architectural governance directly into the development workflow, providing fast feedback to engineers.
5. **Treat Visualization as a First-Class Product:** The ultimate success of the AIM initiative will be determined by its adoption and use by the engineering organization. Therefore, the interactive visualization interface should be treated as a product in its own right, with

a focus on user experience. It must be intuitive, powerful, and tailored to answer the specific architectural questions that are most important to the organization's architects, developers, and technical leaders.

8.3 The Future of Architectural Analysis

The principles and workflow outlined for the Aggregated Interface Map represent a broader shift in the field of software architecture. The era of relying on manually drawn, static diagrams that are outdated the moment they are created is drawing to a close. The future of architectural analysis lies in dynamic, data-driven, and machine-generated models that provide a near real-time reflection of a system's state.

The AIM is a step towards a system of "architectural observability," where the structure, performance, and evolution of the software are continuously monitored and analyzed. As these models become richer and more comprehensive, the next frontier will involve applying machine learning and AI-powered analysis to them.[114] Future systems may be able to automatically detect complex anti-patterns, predict the performance impact of proposed changes, suggest architectural refactorings to reduce coupling or improve resilience, and provide a level of insight into the emergent behavior of complex systems that is currently unattainable. By embarking on the journey to build an AIM, an organization is not just creating a better map of its current systems; it is building the foundation for a more intelligent, data-driven approach to software architecture itself.

Works cited

1. Aggregate Pattern - DevIQ, accessed on September 18, 2025, https://deviq.com/domain-driven-design/aggregate-pattern
2. domain driven design - What is an Aggregate? - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/76373430/what-is-an-aggregate
3. Understanding what really is an aggregate : r/softwarearchitecture - Reddit, accessed on September 18, 2025, https://www.reddit.com/r/softwarearchitecture/comments/1ksw60u/understanding_what_really_is_an_aggregate/
4. architecture - What is aggregates? - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/49476518/what-is-aggregates
5. Application programming interface (API) | EBSCO Research Starters, accessed on September 18, 2025, https://www.ebsco.com/research-starters/computer-science/application-programming-interface-api
6. Understanding "programming to an interface" - Software Engineering Stack

Exchange, accessed on September 18, 2025, https://softwareengineering.stackexchange.com/questions/232359/understanding-programming-to-an-interface
7. Understanding API Data Mapping: A Comprehensive Guide - Adeptia, accessed on September 18, 2025, https://www.adeptia.com/blog/api-data-mapping
8. The Surprise of Multiple Dependency Graphs - ACM Queue, accessed on September 18, 2025, https://queue.acm.org/detail.cfm?id=3723000
9. Application Dependency Mapping: The Complete Guide - Faddom, accessed on September 18, 2025, https://faddom.com/application-dependency-mapping/
10. Towards an Enhanced Dependency Graph - Software REBELs - University of Waterloo, accessed on September 18, 2025, https://rebels.cs.uwaterloo.ca/papers/Meidani_Seyed-Mehran_MMath_202212.pdf
11. Abstract syntax tree - Wikipedia, accessed on September 18, 2025, https://en.wikipedia.org/wiki/Abstract_syntax_tree
12. Abstract syntax trees on Javascript | by Juan Picado - Medium, accessed on September 18, 2025, https://medium.com/@jotadeveloper/abstract-syntax-trees-on-javascript-534e33361fc7
13. A Generic Framework for Automated Quality Assurance of Software Models –Implementation of an Abstract Syntax Tree - Edge Hill University, accessed on September 18, 2025, https://research.edgehill.ac.uk/files/20164772/Paper_5-A_Generic_Framework_for_Automated_Quality_Assurance_of_Software_Models.pdf
14. Static Code Analysis: Everything You Need To Know - Codacy | Blog, accessed on September 18, 2025, https://blog.codacy.com/static-code-analysis
15. Static code analysis: Traversing the AST (Abstract Syntax Tree) provided by Clang through its Python-bindings and building a CFG (Control Flow Graph) and a CG (Call Graph) for the C programming language - shramos's site!, accessed on September 18, 2025, https://www.shramos.com/2018/01/static-code-analysis-traversing-ast.html
16. Static Analysis using ASTs | by Hootsuite Engineering - Medium, accessed on September 18, 2025, https://medium.com/hootsuite-engineering/static-analysis-using-asts-ebcd170c955e
17. Abstract Syntax Trees (ASTs) in Language Processing - Emergent Mind, accessed on September 18, 2025, https://www.emergentmind.com/topics/abstract-syntax-trees-asts
18. What's the use of abstract syntax trees? - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/3860147/whats-the-use-of-abstract-syntax-trees
19. Why do compilers typically convert code into abstract syntax/parse trees before the final product?, accessed on September 18, 2025, https://langdev.stackexchange.com/questions/3662/why-do-compilers-typically-convert-code-into-abstract-syntax-parse-trees-before

20. How to find what is the dependency of a function, class, or variable in ES6 via AST, accessed on September 18, 2025, https://dev.to/jennieji/find-what-is-affected-by-a-declaration-in-javascript-2d5c
21. How do I solve this graphing dependency cycle in an AST?, accessed on September 18, 2025, https://softwareengineering.stackexchange.com/questions/452584/how-do-i-solve-this-graphing-dependency-cycle-in-an-ast
22. AST-Enhanced or AST-Overloaded? The Surprising Impact of Hybrid Graph Representations on Code Clone Detection - arXiv, accessed on September 18, 2025, https://arxiv.org/html/2506.14470v1
23. What is ast-grep?, accessed on September 18, 2025, https://ast-grep.github.io/guide/introduction.html
24. GAST: A Generic AST Representation for Language-Independent Source Code Analysis, accessed on September 18, 2025, https://www.redalyc.org/journal/5722/572276219003/html/
25. Abstract Syntax Tree for Programming Language Understanding and Representation: How Far Are We? - arXiv, accessed on September 18, 2025, https://arxiv.org/html/2312.00413v1
26. ast — Abstract Syntax Trees — Python 3.13.7 documentation, accessed on September 18, 2025, https://docs.python.org/3/library/ast.html
27. extract interface that a class implementing using AST parser - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/15800942/extract-interface-that-a-class-implementing-using-ast-parser
28. How to resolve files with (no ASTs) (failure) ? - Black Duck Community, accessed on September 18, 2025, https://community.blackduck.com/s/article/How-to-resolve-files-with-no-ASTs-failure
29. How to extract all the dependencies from the pom.xml? - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/53282487/how-to-extract-all-the-dependencies-from-the-pom-xml
30. Converting Gradle Build File to Maven POM | Baeldung, accessed on September 18, 2025, https://www.baeldung.com/gradle-build-to-maven-pom
31. Running Gradle inside Maven - Andres Almiray, accessed on September 18, 2025, https://andresalmiray.com/running-gradle-inside-maven/
32. Is this the best way to access the pom file for a dependency? - Gradle Forums, accessed on September 18, 2025, https://discuss.gradle.org/t/is-this-the-best-way-to-access-the-pom-file-for-a-dependency/5409
33. How to import dependecies from build.gradle to pom.xml - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/34484733/how-to-import-dependecies-from-build-gradle-to-pom-xml
34. package.json - npm Docs, accessed on September 18, 2025,

https://docs.npmjs.com/files/package.json/
35. Specifying dependencies and devDependencies in a package.json file - npm Docs, accessed on September 18, 2025, https://docs.npmjs.com/specifying-dependencies-and-devdependencies-in-a-package-json-file/
36. A Developer's Tutorial to Using NPM Audit for Dependency Scanning - Spectral, accessed on September 18, 2025, https://spectralops.io/blog/a-developers-tutorial-to-using-npm-audit-for-dependency-scanning/
37. Understanding dependencies inside your Package.json - NodeSource, accessed on September 18, 2025, https://nodesource.com/blog/understanding-dependencies-inside-your-package json
38. A Deep Dive into package.json: Understanding Every Dependency Type - DEV Community, accessed on September 18, 2025, https://dev.to/mechcloud_academy/a-deep-dive-into-packagejson-understanding-every-dependency-type-2jm2
39. What Is OpenAPI? | Swagger Docs, accessed on September 18, 2025, https://swagger.io/docs/specification/v3_0/about/
40. OpenAPI Specification - Version 3.1.0 - Swagger, accessed on September 18, 2025, https://swagger.io/specification/
41. OpenAPI 3.0 Tutorial: OpenAPI Specification Definition - Apidog, accessed on September 18, 2025, https://apidog.com/blog/openapi-specification/
42. Guide to Swagger Parser | Baeldung, accessed on September 18, 2025, https://www.baeldung.com/java-swagger-parser
43. OpenAPI Design & Documentation Tools - Swagger, accessed on September 18, 2025, https://swagger.io/tools/open-source/
44. OpenAPI tools - Agent Development Kit - Google, accessed on September 18, 2025, https://google.github.io/adk-docs/tools/openapi-tools/
45. 2.2. Command Line Usage: scan-build and CodeChecker - Clang Static Analyzer - LLVM, accessed on September 18, 2025, https://clang-analyzer.llvm.org/scan-build.html
46. Configure code analysis rules - .NET - Microsoft Learn, accessed on September 18, 2025, https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/configuration-options
47. What is Bytecode Analysis? - Cloudmersive APIs, accessed on September 18, 2025, https://cloudmersive.com/article/What-is-Bytecode-Analysis%3F
48. DALEQ -- Explainable Equivalence for Java Bytecode - arXiv, accessed on September 18, 2025, https://www.arxiv.org/pdf/2508.01530
49. The feature extraction of Java bytecode | Download Scientific Diagram - ResearchGate, accessed on September 18, 2025, https://www.researchgate.net/figure/The-feature-extraction-of-Java-bytecode_fig4_330984458
50. DALEQ - Explainable Equivalence for Java Bytecode - arXiv, accessed on

September 18, 2025, https://arxiv.org/html/2508.01530v1
51. Coffea - Static dependency analyzer for Java bytecode. - GitHub, accessed on September 18, 2025, https://github.com/sjednac/coffea
52. What are some useful static analyzers for Java? - Reddit, accessed on September 18, 2025, https://www.reddit.com/r/java/comments/rur1yi/what_are_some_useful_static_analyzers_for_java/
53. Compiled Python Files - Phylum.io, accessed on September 18, 2025, https://blog.phylum.io/compiled-python-files/
54. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology | PLOS One - Research journals, accessed on September 18, 2025, https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0225196
55. ANTLR Tree Construction, accessed on September 18, 2025, https://www.antlr2.org/doc/trees.html
56. Creating a simple parser with ANTLR - Ivan Yurchenko, accessed on September 18, 2025, https://ivanyu.me/blog/2014/09/13/creating-a-simple-parser-with-antlr/
57. Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection - Yueming Wu, accessed on September 18, 2025, https://wu-yueming.github.io/Files/ICSE2024_Toma.pdf
58. Best Practices for ANTLR Parsers - Strumenta - Federico Tomassetti, accessed on September 18, 2025, https://tomassetti.me/best-practices-for-antlr-parsers/
59. ANTLR: exclude (skip) tokens when building AST tree - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/57432736/antlr-exclude-skip-tokens-when-building-ast-tree
60. [1301.0849] Static Analysis for Regular Expression Denial-of-Service Attacks - arXiv, accessed on September 18, 2025, https://arxiv.org/abs/1301.0849
61. Using Static Code Analysis to Improve Log Parsing - Palantir Blog, accessed on September 18, 2025, https://blog.palantir.com/using-static-code-analysis-to-improve-log-parsing-18f0d1843965
62. Limitations of Regular Expressions? - regex - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/7854063/limitations-of-regular-expressions
63. Why it's not possible to use regex to parse HTML/XML: a formal explanation in layman's terms [duplicate] - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/6751105/why-its-not-possible-to-use-regex-to-parse-html-xml-a-formal-explanation-in-la
64. ELI5- Why can't regex parse HTML? : r/AskProgramming - Reddit, accessed on September 18, 2025, https://www.reddit.com/r/AskProgramming/comments/12k2t02/eli5_why_cant_regex_parse_html/
65. Static Code Analysis Tools: Beyond the Regex with CodeQL | Abdul Wahab Junaid, accessed on September 18, 2025, https://awjunaid.com/books/static-code-analysis-tools-beyond-the-regex-with-c

odeql/
66. regular expressions to analyse source code - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/2005751/regular-expressions-to-analyse-source-code
67. Static analysis and regular expressions - PVS-Studio, accessed on September 18, 2025, https://pvs-studio.com/en/blog/posts/cpp/0087/
68. When you should NOT use Regular Expressions? - Software Engineering Stack Exchange, accessed on September 18, 2025, https://softwareengineering.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions
69. Regexes are Hard: Decision-making, Difficulties, and Risks in Programming Regular Expressions - Francisco Servant, accessed on September 18, 2025, https://fservant.github.io/papers/Michael_Donohue_Davis_Lee_Servant_ASE19.pdf
70. Static Code Analysis - Wiz, accessed on September 18, 2025, https://www.wiz.io/academy/static-code-analysis
71. View the change history of a file using Git versioning - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/278192/view-the-change-history-of-a-file-using-git-versioning
72. API Versioning while maintaining git history - python - Stack Overflow, accessed on September 18, 2025, https://stackoverflow.com/questions/14271489/api-versioning-while-maintaining-git-history
73. Mastering Git for API Version Management to Ensure Seamless Integrations - APIPark, accessed on September 18, 2025, https://apipark.com/technews/YFyPlRpC.html
74. REST API endpoints for dependency review - GitHub Docs, accessed on September 18, 2025, https://docs.github.com/en/rest/dependency-graph/dependency-review
75. How to navigate API evolution with versioning - Red Hat Developer, accessed on September 18, 2025, https://developers.redhat.com/articles/2024/03/25/how-navigate-api-evolution-versioning
76. Data mining historical insights for a software keyword from GitHub and Libraries.io; GraphQL - DiVA portal, accessed on September 18, 2025, https://www.diva-portal.org/smash/get/diva2:1675966/FULLTEXT01.pdf
77. Simplifying Debugging in Distributed Systems with Tracing API Calls - Ambassador Labs, accessed on September 18, 2025, https://www.getambassador.io/blog/tracing-api-calls-debugging-distributed-systems
78. Tracing API - OpenTelemetry, accessed on September 18, 2025, https://opentelemetry.io/docs/specs/otel/trace/api/
79. What is Distributed Tracing? Concepts & OpenTelemetry Implementation |

Uptrace, accessed on September 18, 2025, https://uptrace.dev/opentelemetry/distributed-tracing
80. Top APM Tools for Java in 2025: JVM Metrics, Spring Boot Tracing & Cost at Scale, accessed on September 18, 2025, https://cubeapm.com/blog/top-apm-tools-for-java/
81. 17 Best Java Application Monitoring Tools in 2025 - Sematext, accessed on September 18, 2025, https://sematext.com/blog/java-monitoring-tools/
82. Dependency Tracking in Application Insights - Azure Monitor | Microsoft Learn, accessed on September 18, 2025, https://learn.microsoft.com/en-us/azure/azure-monitor/app/dependencies
83. Essential Guide to Microservices Monitoring in 2025 - SigNoz, accessed on September 18, 2025, https://signoz.io/guides/microservices-monitoring/
84. Mastering Microservices Monitoring: Best Practices and Tools - Middleware, accessed on September 18, 2025, https://middleware.io/blog/microservices-monitoring/
85. Distributed Tracing 101: Definition, Working and Implementation | Last9, accessed on September 18, 2025, https://last9.io/blog/challenges-of-distributed-tracing/
86. Frida • A world-class dynamic instrumentation toolkit | Observe and reprogram running programs on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX, accessed on September 18, 2025, https://frida.re/
87. Dynamic Analysis and Runtime Manipulation in Mobile Pentesting | by Izaz Haque | Medium, accessed on September 18, 2025, https://medium.com/@izaz.haque246/dynamic-analysis-and-runtime-manipulation-in-mobile-pentesting-b9e991344bb9
88. Dynamic analysis and tampering - Licel, accessed on September 18, 2025, https://licelus.com/resources/guide-to-mobile-application-protection/threats/dynamic-analysis-and-tampering
89. Guide to Java Instrumentation | Baeldung, accessed on September 18, 2025, https://www.baeldung.com/java-instrumentation
90. Java SE Tools - Oracle, accessed on September 18, 2025, https://www.oracle.com/java/technologies/javase/tools-jsp.html
91. Microservices Monitoring Strategies and Best Practices - Catchpoint, accessed on September 18, 2025, https://www.catchpoint.com/api-monitoring-tools/microservices-monitoring
92. eBPF Explained: Use Cases, Concepts, and Architecture | Tigera - Creator of Calico, accessed on September 18, 2025, https://www.tigera.io/learn/guides/ebpf/
93. Network Monitoring - Datadog, accessed on September 18, 2025, https://www.datadoghq.com/product/network-monitoring/
94. Network monitoring - Dynatrace, accessed on September 18, 2025, https://www.dynatrace.com/platform/network-monitoring/
95. Microservices: Service Discovery Patterns and 3 Ways to Implement - Solo.io, accessed on September 18, 2025, https://www.solo.io/topics/microservices/microservices-service-discovery
96. Service Discovery in Microservices: A Detailed Guide | by Dev Cookies | Medium, accessed on September 18, 2025,

https://devcookies.medium.com/service-discovery-in-microservices-a-detailed-guide-dc5184777508
97. Service Discovery in Microservices | Baeldung on Computer Science, accessed on September 18, 2025, https://www.baeldung.com/cs/service-discovery-microservices
98. Understanding Service Discovery in Microservices Architecture | by Jeslur Rahman, accessed on September 18, 2025, https://medium.com/@jeslurrahman/understanding-service-discovery-in-microservices-architecture-2098b10e7439
99. Microservice Service Discovery: API Gateway vs Service Mesh? - Ambassador Labs, accessed on September 18, 2025, https://www.getambassador.io/blog/microservices-discovery-api-gateway-vs-service-mesh
100. Service Discovery in Microservices: Key Insights - Edge Delta, accessed on September 18, 2025, https://edgedelta.com/company/blog/what-is-service-discovery
101. What is Blind Spot? | Netenrich Fundamentals, accessed on September 18, 2025, https://netenrich.com/fundamentals/blind-spot
102. 10 Most Common Cybersecurity Blind Spots - Balbix, accessed on September 18, 2025, https://www.balbix.com/insights/10-blindspots-in-your-cybersecurity-posture/
103. What is Dynamic Application Security Testing (DAST)? Examples & Features - Apiiro, accessed on September 18, 2025, https://apiiro.com/glossary/dynamic-application-security-testing/
104. Static vs. dynamic code analysis: A comprehensive guide - vFunction, accessed on September 18, 2025, https://vfunction.com/blog/static-vs-dynamic-code-analysis/
105. Static vs. Dynamic Code Analysis for Clinical Apps - Censinet, accessed on September 18, 2025, https://www.censinet.com/perspectives/static-vs-dynamic-code-analysis-for-clinical-apps
106. Static vs. Dynamic Code Analysis: How to Choose Between Them - Harness, accessed on September 18, 2025, https://www.harness.io/blog/static-vs-dynamic-code-analysis
107. Static vs. Dynamic Code Analysis - Qt, accessed on September 18, 2025, https://www.qt.io/quality-assurance/blog/static-vs-dynamic-code-analysis
108. Dynamic Testing vs. Static Testing: Pros and Cons Revealed - Prometteur Solutions, accessed on September 18, 2025, https://prometteursolutions.com/blog/dynamic-testing-vs-static-testing-pros-and-cons-revealed/
109. Difference Between Dynamic and Static Testing: Pros and Cons - Testbytes, accessed on September 18, 2025, https://www.testbytes.net/blog/dynamic-testing-and-static-testing/
110. Static vs. dynamic code analysis: advantages and disadvantages - Route Fifty, accessed on September 18, 2025,

https://www.route-fifty.com/cybersecurity/2009/02/static-vs-dynamic-code-analysis-advantages-and-disadvantages/287891/
111. OpenAPI Specification Discovery — ThreatNG Security - External Attack Surface Management (EASM) - Digital Risk Protection, accessed on September 18, 2025, https://www.threatngsecurity.com/glossary/openapi-specification-discovery
112. A Practical Guide to Application Security Testing: Methods, Tools, and Real-World Integration, accessed on September 18, 2025, https://www.ox.security/blog/application-security-testing/
113. Static Code Analysis Best Practices for Developers - ACCELQ, accessed on September 18, 2025, https://www.accelq.com/blog/static-code-analysis-best-practices/
114. How Dependency Visualization Revolutionizes Software Architecture Design - ONES.com, accessed on September 18, 2025, https://ones.com/blog/dependency-visualization-revolutionizes-software-architecture-design/
115. Dependency Graph Visualization - Tom Sawyer Software - Blog, accessed on September 18, 2025, https://blog.tomsawyer.com/dependency-graph-visualization
116. On the Role of Metaphor in Information Visualization - arXiv, accessed on September 18, 2025, https://arxiv.org/pdf/0809.0884
117. A Layered Software City for Dependency Visualization - SciTePress, accessed on September 18, 2025, https://www.scitepress.org/Papers/2021/101802/101802.pdf
118. Metaphor Representation and Analysis of Non-Spatial Data in Map-Like Visualizations, accessed on September 18, 2025, https://doaj.org/article/2b91745dd5bd46b3aceb51249588f872
119. Information Cartography: Using GIS for Visualizing Non-spatial Data - Recent Proceedings, accessed on September 18, 2025, https://proceedings.esri.com/library/userconf/proc02/pap0239/p0239.htm
120. (PDF) Visualization Metaphors - ResearchGate, accessed on September 18, 2025, https://www.researchgate.net/publication/225203640_Visualization_Metaphors
121. Biological Metaphors in he Design of Complex Software Systems - Purdue e-Pubs, accessed on September 18, 2025, https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2448&context=cstech
122. Structurizr, accessed on September 18, 2025, https://structurizr.com/
123. Gephi - The Open Graph Viz Platform, accessed on September 18, 2025, https://gephi.org/
124. NetworkX — NetworkX documentation, accessed on September 18, 2025, https://networkx.org/
125. D3 by Observable | The JavaScript library for bespoke data visualization, accessed on September 18, 2025, https://d3js.org/