MASARYK UNIVERSITY

FACULTY OF INFORMATICS

LLM-assisted Software Architecture Analysis

Master's Thesis

PETER VERČIMÁK

Brno, Spring 2026

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

LLM-assisted Software Architecture Analysis

Master's Thesis

PETER VERČIMÁK

Advisor: doc. Bruno Rossi, PhD.

Department of Computer Systems and Communications

Brno, Spring 2026

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI-assisted tools:

- Grammarly for grammar checking, language checking, and wording suggestions.
- Claude Code and OpenAI Codex for code-related assistance, code generation, implementation discussion, and thesis-structure discussion.

I declare that I used these tools in accordance with the principles of academic integrity. I reviewed and verified their outputs, and I take full responsibility for the final content of this thesis.

Peter Verčimák

Advisor: doc. Bruno Rossi, PhD.

Acknowledgements

I would like to thank my supervisor, doc. Bruno Rossi, PhD., for his guidance on the topic and direction of this thesis and for his patient answers to my questions throughout the work.

I am also grateful to my external consultant, Ing. Lukáš Hübner, for his technical advice during the implementation of the system. His perspective on practical engineering decisions was invaluable.

Abstract

This thesis presents a tool for static architectural analysis of C# solutions combined with explanation by a local large language model. Roslyn is used to extract a typed dependency graph that records signature-level and body-level relationships between classes, interfaces, structs, and records, with each edge labelled by its dependency kind and backed by source evidence. A family of graph analyzers detects cyclic dependencies, hub-like coupling, god-class candidates, and blueprint-driven layer violations under a shared finding contract. A reporting pipeline transforms findings into structured packets containing extracted facts, interpretations, and a fixed set of remediation strategies. A locally hosted language model reads these packets and presents them conversationally without authoring architectural facts of its own. The delivery layer exposes the pipeline through an HTTP API, a Model Context Protocol tool surface, and an interactive command-line bridge with retry guards that catch common grounding failures. The solution is evaluated on a synthetic stress-lab solution, a self-analysis of the tool's own codebase, and two .NET course projects from the Faculty of Informatics at Masaryk University. The evaluation also examines the effect of local model size and context-window capacity on the quality of the produced explanations.

Keywords

LLM-assisted software engineering, software architecture analysis, static analysis, dependency graph, architectural smells, C#, Roslyn, grounded reporting

Contents

1 Introduction **1**
- 1.1 Context and Motivation 1
- 1.2 Problem Statement 1
- 1.3 Thesis Objectives and Scope 2
- 1.4 Contributions 3
- 1.5 Thesis Structure 4

2 Background **5**
- 2.1 Static Analysis of C# Programs 5
- 2.2 Dependency Graphs and Architectural Smells 6
- 2.3 Roslyn-Based Program Analysis 7
- 2.4 LLM-Assisted Software Engineering and Grounding . . 9

3 Related Work **11**
- 3.1 Static Architecture Analysis Tools 11
- 3.2 Language Models as Tool-Using Assistants 12
- 3.3 Positioning of DotArchSentry 13

4 Solution Architecture **15**
- 4.1 Overview and Main Use Cases 15
- 4.2 End-to-End Pipeline 16
- 4.3 Subsystems and Architectural Boundaries 17
- 4.4 Configuration, Blueprints, and Persistence 20

5 Dependency Extraction and Graph Model **22**
- 5.1 Input Scope and Extraction Context 22
- 5.2 Graph Construction Pipeline 23
- 5.3 Extraction Outputs: Type Graph and Project Snapshot . 25
- 5.4 Signature and Body-Level Dependency Extraction . . . 26
- 5.5 Evidence, Enrichment, and Graph Representation . . . 28
- 5.6 Extraction Limitations 30

6 Architectural Analysis and Finding Artifacts **32**
- 6.1 Analyzer Pipeline and Sensitivity Settings 32
- 6.2 Architecture Blueprints 34
- 6.3 Detected Architectural Issues 36

6.4 Finding Model, Metrics, and Severity 38

7 Reporting Pipeline **42**
7.1 Consumers of Analysis Output 42
7.2 Finding-to-Packet Transformation 42
7.3 Inspection and Recommendation Packets 44
7.4 Run Comparison and Architectural Drift 45

8 LLM Integration and Grounded Delivery **49**
8.1 Role and Boundaries of the LLM 49
8.2 Grounded Briefs and Packet Presentation 50
8.3 System Prompt, Tool Routing, and Context Management 52
8.4 Grounding Controls and Retry Gates 53

9 Evaluation **56**
9.1 Evaluation Design 56
9.1.1 Corpus 56
9.1.2 Run Conditions 56
9.2 Static Architecture Analysis Results 57
9.2.1 StressLab: Designed Findings and Run Comparison 57
9.2.2 Self-Analysis and NDepend Cross-Check 58
9.2.3 External Course Projects 58
9.3 LLM-Assisted Interpretation Results 59
9.3.1 Harness, Rubric, and Manual Semantic Audit . 59
9.3.2 Results at the 4096-Token Default 60
9.3.3 The 2048-Token Stress Point 60
9.3.4 Observed Failure Modes 61
9.4 Threats to Validity 61

10 Conclusion **63**
10.1 Summary of Contributions 63
10.2 Limitations and Future Work 64

Bibliography **65**

A Evaluation Artifacts **69**

B Running the Submitted Application **71**

List of Tables

9.1 Cross-tool comparison on the DotArchSentry self-analysis codebase. 58
9.2 Manual audit results at the 2048-token stress point, aggregated across the three model sizes. 60

List of Figures

4.1 High-level DotArchSentry pipeline from solution input to reporting and delivery. 16
4.2 DotArchSentry project boundaries and dependency direction. Solid arrows denote project references; dashed arrows denote runtime communication. 19
5.1 Dependency extraction paths. The type-level dependency graph is the main extraction output. A separate project-dependency snapshot records declared project references for layer analysis at project scope. 24
6.1 Analyzer pipeline from analysis inputs to enriched and scored finding artifacts. 33
7.1 Finding-to-packet transformation for inspection and recommendation workflows. 43
7.2 Run-comparison workflow from compatibility checking to ordered finding diff. 46

1 Introduction

1.1 Context and Motivation

Modern software systems are rarely static. They are continuously extended, adapted, and maintained over long periods of time, often by more than one development team. As systems evolve, their code structure tends to become increasingly complex, making it harder to preserve the original architectural intent. This phenomenon is commonly referred to as architectural erosion [1].

One of the biggest contributors to architectural complexity is the structure of dependencies within a system. Dependencies between software components determine how changes propagate, how easily parts of the system can be understood in isolation, and how robust the architecture is against future additions or changes. While dependencies are unavoidable in non-trivial systems, their uncontrolled growth can lead to fragile architectures.

In practice, developers and architects often rely on informal knowledge or partial tooling support to understand architectural dependencies. This becomes increasingly problematic as systems grow in size and age, where challenges such as legacy code, time pressure, and team turnover are common. Automated support for analyzing and understanding architectural dependencies is, therefore, an important prerequisite for maintaining software quality over time.

1.2 Problem Statement

Despite the recognized importance of architectural dependencies and the need for automated support, it is still difficult to analyze them within practical, real-world software systems. Large codebases may contain thousands of types and an even larger number of dependency relationships, making manual inspection barely feasible and very time consuming. As a result, architectural problems like cyclic dependencies, tight coupling, or violations of intended layering tend to appear gradually and frequently go unnoticed until they seriously obstruct maintenance and ongoing development.

Static analysis can extract dependency information directly from the source code and is commonly used in both research and practice. In many cases, however, the output is either too low-level or too fragmented. For example, tools report individual warnings or metrics, but do not provide a clear architectural view. As a result, it is often difficult to understand how the reported issues relate to the overall structure of the system and what they imply at the architectural level.

Research in areas such as dependency analysis, architecture recovery, and software metrics has addressed the extraction and representation of software structure. These studies provide valuable foundations for architectural analysis, yet the interpretation of analysis results is still largely left to human experts. This reliance on manual interpretation limits scalability and practical applicability, particularly in large or evolving systems. The problem addressed by this thesis is how architectural dependencies in C# software systems can be analyzed and presented in a way that supports both precise detection of architectural issues and meaningful, explainable interpretation of the results.

1.3 Thesis Objectives and Scope

To address the problem stated in the previous section, this thesis aims to support architectural reasoning in C# software systems through dependency extraction, graph-based analysis, and explainable outputs. The work is organized around the following goals:

- Statically extract dependency information from a C# solution (given a path to a `.sln`) using Roslyn, capturing relationships between projects, namespaces, and types.
- Represent the extracted relationships as a typed dependency graph, with nodes corresponding to architectural elements and edges capturing concrete dependency kinds, such as type usage via fields or properties, method parameters and return types, invocations, member access, and object creation.
- Implement a suite of graph-based architectural analyses over the extracted dependency graph to identify maintainability-relevant

structural issues (cyclic dependencies, coupling/hub detection, layering or forbidden-dependency violations, and "god class" indicators).

- Design a reporting pipeline that transforms findings into typed packet artifacts through explicit fact extraction, interpretation, and strategy-selection stages, and build a delivery layer over this pipeline (HTTP API, MCP server, and a local Ollama bridge) in which a large language model presents the packets to users while remaining bounded to the supplied evidence.

- Evaluate the solution on selected C# systems, compare selected findings with existing analyzers or tools, and examine how local LLM model size and context-window capacity affect the quality of generated explanations and recommendations.

1.4 Contributions

- a Roslyn-based extractor that produces a directed dependency graph in which each edge is labelled with one of seventeen dependency kinds and carries the source file, line, snippet, and enclosing member of every observation that contributed to it,

- a graph-analyzer family that detects type-level cycles, hub-like coupling, and blueprint-driven layer violations at namespace and project scope, sharing one finding-artifact contract,

- a reporting pipeline that converts each finding into a small packet (a fact record, an interpretation, and, for recommendations, a fixed set of remediation strategies) read by a small local LLM that presents the packet without authoring its own findings or strategy choices,

- a delivery layer exposing the pipeline as an HTTP API, an MCP-tool surface, and an interactive CLI driven by a local Ollama model, with no cloud calls in the analysis or reporting paths.

1.5 Thesis Structure

The remainder of the thesis is organized as follows.

- Chapter 2 introduces the technical and conceptual foundations required for the thesis.
- Chapter 3 discusses related work and positions DotArchSentry against existing tools and research.
- Chapter 4 presents the overall architecture of DotArchSentry.
- Chapter 5 describes dependency extraction and the dependency graph model.
- Chapter 6 presents the architectural analysis layer and the produced finding artifacts.
- Chapter 7 describes the reporting pipeline that turns findings into structured packets and supports run comparison.
- Chapter 8 describes how the LLM interprets and presents these packets: the brief that prepares each packet for the model, the system prompt that routes user intent into tool calls, and the retry gates that catch residual hallucinations.
- Chapter 9 evaluates the solution on selected systems and examines how LLM model size affects interpretation quality.
- Chapter 10 concludes the thesis and outlines limitations and future work.

2 Background

This chapter defines the concepts used by the thesis: source-level static analysis, dependency graphs, Roslyn program analysis, and the use of language models with externally supplied evidence.

2.1 Static Analysis of C# Programs

Static analysis examines a program without executing it. For C# systems, the analyzed material usually consists of source files, project files, references, and the compiler view built from them. Static analysis can operate at several representation levels: source text, parsed syntax, and compiler-resolved semantics. Architectural dependency extraction depends on the latter two levels. Syntax provides source structure and locations, while semantics identifies the program entities involved.

Architectural static analysis differs from ordinary compiler diagnostics. A compiler diagnostic is usually tied to one local program error or warning. Architectural analysis looks for structural patterns distributed across a solution. A dependency cycle, a central coupling point, or an unexpected layer crossing may involve several files and projects. The relevant object is therefore the structure of the codebase, not only the correctness of one statement.

The strength of static analysis is systematic coverage. The same extraction rule can be applied to every project in a solution. If a field declaration, method signature, inheritance relationship, or object creation creates a dependency, the analyzer can record it consistently across the codebase. This makes static analysis suitable for building a structural model that would be tedious to maintain manually.

The limits are equally important. Static analysis is strongest for relationships represented in source code and resolved by the compiler. Runtime mechanisms can escape this view. Reflection, dependency-injection configuration, plugin loading, generated code, and external configuration files may create dependencies that are real at runtime but weakly represented in ordinary source references. A static architecture tool therefore provides source-derived structural evidence. It should not be read as a complete runtime trace.

2.2 Dependency Graphs and Architectural Smells

A dependency graph represents software structure as nodes and directed edges. In architectural analysis, nodes often stand for projects, namespaces, or types. A directed edge from one node to another records that the source node depends on the target node. The edge may also carry the dependency category, the number of observations, and the source location where the relationship was found.

Graphs fit architectural reasoning because many design problems are relational. A type can become risky because many other types depend on it. A namespace boundary can weaken because dependencies repeatedly cross it. A group of classes can become difficult to change because the group contains a directed cycle. In all three cases, the issue is not visible from one node in isolation. The position of the node inside the graph matters.

Architectural erosion is the gradual loss of intended structure as a software system evolves [1]. Dependency graphs make some forms of erosion observable. If a system is expected to follow layers, forbidden edges show where that rule is weakening. If a module is expected to remain independent, incoming and outgoing edges show whether that independence still holds.

Cyclic dependencies are a common graph-level smell. A cycle means that a set of elements depends on itself through a directed path. At type level, cycles complicate ownership, construction, and change order. Strongly connected components provide the graph concept used to detect such structures. In a non-trivial strongly connected component, every member is reachable from the others through directed paths. The two-pass algorithm used for this kind of detection is commonly referred to as the Kosaraju–Sharir algorithm, with Sharir's 1981 paper serving as the published reference [2].

Hub-like coupling is another recurring graph pattern. High incoming dependency count can indicate a central change target. High outgoing dependency count can indicate a type that coordinates too many collaborators.

Some architectural smells combine graph structure with object-oriented metrics. God-class detection is a typical example. Metric-based detection strategies use several measurements together, including size, responsibility, and coupling [3, 4]. Coupling measurement

itself is also multi-dimensional. Object-oriented coupling is better understood as a family of relationships with different meanings and strengths [5].

Layer violations need one extra ingredient: an architectural policy. A graph can show that one element depends on another. It cannot decide whether that relationship is allowed unless a rule set exists. Layer analysis therefore combines observed dependencies with a policy that states which dependency directions are permitted.

2.3 Roslyn-Based Program Analysis

Roslyn is the .NET Compiler Platform. It exposes the C# compiler through APIs for syntax trees, semantic models, symbols, compilations, and workspaces [6]. These APIs make it possible to build source-code analysis tools using compiler information rather than plain text matching.

A syntax tree represents the structure of a source file [7]. Its nodes correspond to language constructs. A class declaration contains members. A method declaration contains a body. A body contains statements and expressions. This tree is close to what is usually called an abstract syntax tree, although Roslyn keeps source-level detail needed by tooling, including trivia, formatting, and source spans.

The tree form matters because it gives analyzers precise locations. If an object creation expression appears inside a method, the syntax node carries a span in the source file. If a field declaration introduces a stored reference, the declaration node can be mapped back to a line. A static architecture tool can therefore attach source locations and snippets to extracted dependencies instead of reporting only that a dependency exists.

Syntax describes the structural form of source code, such as declarations, expressions, method calls, object creations, and type names. However, architectural dependency extraction also requires semantic information, because the textual form of the code does not always determine which program entity is being referenced. The same name can refer to different symbols in different scopes. Similarly, a call expression may target a method declared on a concrete class, an interface, or a base type. A type name may refer to a type defined in the current

project, another project, or an external assembly. Therefore, text alone cannot distinguish these cases reliably.

Roslyn provides semantic information through the semantic model [8]. The semantic model connects syntax nodes from the syntax tree with compiler symbols. Given an invocation expression, it can identify the method that is actually being called. Given an object creation expression, it can identify the constructed type. Given a field or property declaration, it can identify the declared type. These resolutions turn syntactic observations into precise dependency facts.

Symbols are Roslyn's representation of program entities. A named type symbol represents a class, interface, struct, record, or enum. Other symbol APIs expose information about methods, properties, fields, namespaces, assemblies, parameters, and local variables. Stable symbol identity is important for architecture analysis because the same type can appear in source code through different textual forms, such as aliases, using directives, or fully qualified names. Therefore, the analyzer needs the resolved compiler entity, not only the text that appeared in the source file.

A compilation represents the compiler view of one project. It combines syntax trees with references, parse options, compilation options, and metadata. Semantic resolution depends on this compilation. If a project reference or metadata reference is missing, the semantic model may still parse the source, but some symbols will be unresolved. Correct project loading is therefore part of analysis precision.

The workspace model raises the view from a single project to a solution [9]. A workspace contains projects, documents, project references, metadata references, and solution-level structure. This is the level needed for architectural analysis of C# solutions. Project membership and project references are architectural facts in their own right, and they also determine which symbols can be resolved inside each project.

Roslyn analysis therefore works at three connected levels. Syntax trees provide source structure and locations. Semantic models provide resolved meaning. Workspaces provide solution context. A dependency extractor for C# architecture needs all three.

2.4 LLM-Assisted Software Engineering and Grounding

Large language models are commonly used as assistants for software engineering tasks. At the level needed here, an LLM is a model that generates text from a prompt and previous context. For chat models, the context can include the system instruction, recent conversation history, tool outputs, and structured data inserted by the application. The model generates the next answer from this combined context, so factual support has to come from the information supplied to it.

Assistant-style interaction usually relies on chat-oriented models trained to respond to user requests. Model size is one studied factor in their task and prompting performance [10]. For local software-engineering tools, size is also a deployment constraint. Larger models need more memory and compute, while smaller models are easier to run interactively on ordinary developer hardware.

Ollama is one practical way to run such models locally [11]. It manages downloaded model tags and exposes a local service interface. Local execution is useful for source-code tools because prompts may contain project names, dependency evidence, source snippets, or architectural findings that should remain on the user's machine. It also makes the concrete model tag part of the experimental setup. A result obtained with one locally installed model is therefore tied to that model tag and runtime setup.

Generated answers can be fluent while unsupported by the supplied input. In natural language generation, hallucination refers to generated content that is unsupported by, or inconsistent with, the source material [12]. Summarization research uses the related concept of faithfulness to describe whether generated text stays true to the input document [13]. Software explanations have the same risk. A model can name a dependency that was not extracted, attribute evidence to the wrong class, or suggest a refactoring strategy that was never supplied.

In this thesis, grounding means that generated explanations are constrained by an external source of facts. The source can be a finding packet, source evidence, or a fixed set of allowed strategies. The model may phrase and organize those facts, but the facts themselves come

from the analysis output. This idea is related to retrieval-augmented generation, where generated text is conditioned on retrieved external material rather than only on model parameters [14]. Here, the retrieved material is structured analysis output rather than a document collection.

Input shape is part of the problem. Language models do not always use all positions in a long context equally well, and information placed in the middle of the input can be missed [15]. A raw JSON packet can therefore contain the necessary field while the model still fails to use it. For LLM-assisted architecture tools, source-supported generation depends on the available facts and on how those facts are presented.

3 Related Work

This chapter places DotArchSentry among existing tools and approaches. The first section surveys static architecture-analysis tools and architecture-rule libraries. The second section covers LLM assistants that use tools or external context. The final section positions DotArchSentry against those areas and states the specific gap addressed by the thesis.

3.1 Static Architecture Analysis Tools

The tools selected here cover four common workflows around architectural analysis: SonarQube for general static quality control, NDepend for .NET dependency and metric inspection, ArchUnitNET and NetArchTest for architecture rules as tests, and Arcan for architecture-smell detection from dependency models.

SonarQube. SonarQube is a hosted multi-language analyzer that runs in continuous integration and posts results to a server as line-anchored issues, project-level measures, and quality-gate decisions [16]. A developer's interaction is with individual issues: triage, suppression, code fix, or an accepted gate failure. SonarQube's coverage spans reliability, security, duplication, coverage, and maintainability; architectural concerns sit inside that wider model rather than as a first-class structural surface. The point of comparison for this thesis is the delivery model: static-analysis findings presented to development teams through an issue-and-gate workflow, with architecture treated as one category among many.

NDepend. NDepend is a .NET architecture-analysis tool that presents a codebase through a dependency matrix, dependency graphs, code-metric tables, and CQLinq queries, which are rule expressions written in a C#-like syntax [17]. Dependency structure is a first-class object in the tool: the matrix and graph are central inspection views, and CQLinq lets users encode project-specific rules over the analyzed codebase. Trend monitoring across runs supports treating architectural findings as data that changes over time. Among the tools surveyed in this section, NDepend is the closest .NET-specific commercial product for dependency-centered architecture analysis.

ArchUnitNET and NetArchTest. Architecture test libraries encode architectural expectations as executable checks. ArchUnitNET provides rules over assemblies, namespaces, types, dependencies, slices, and cycles, and can run those checks through common .NET test frameworks [18]. NetArchTest is a lighter fluent-rule library for selecting .NET types from assemblies or namespaces and asserting naming, design, or dependency constraints [19]. Both fit teams that already know the rule they want to enforce. A team can encode, for example, that one namespace must not depend on another or that selected classes must follow a naming convention. Unknown smells still require a rule or test to be written before the tool can report them.

Arcan. Arcan is the closest related tool in the architecture-smell category. It analyzes dependency information and reports smells such as cyclic dependency, hub-like dependency, god component, and unstable dependency [20]. Its documentation describes cycle detection over dependency graphs, hub-like dependency through high incoming and outgoing dependency counts, and unstable dependency through dependencies toward less stable components [21]. Arcan is therefore a dedicated smell detector rather than a general quality dashboard or a rule-testing library. It is the main related-work reference for graph-based cycle and hub analysis in this thesis.

Together, these tools establish the surrounding landscape. Quality platforms support project governance. Dependency tools support structural inspection. Architecture-test libraries enforce known constraints. Smell detectors search dependency models for recurring design problems. DotArchSentry is compared against these categories in Section 3.3.

3.2 Language Models as Tool-Using Assistants

LLM assistants are increasingly used beyond plain text generation. In a tool-using setting, the model does not answer only from its parameters and prompt. It can choose an external action, receive the result, and continue the conversation using that result. This pattern matters for software-engineering tools because many relevant facts are held by external systems: compilers, test runners, static analyzers, package managers, issue trackers, and project repositories.

Research on reasoning-and-acting agents studies this combination directly. ReAct interleaves language-model reasoning with actions that query external environments or knowledge sources [22]. Toolformer studies models that learn to decide when to call external APIs, what arguments to pass, and how to use the returned result [23]. These works are broader than architecture analysis, but they capture the same design shift: an LLM can be treated as a controller over tools rather than as the sole source of facts.

The Model Context Protocol (MCP) is a practical interface for this style of integration. It defines a way for applications to expose tools and context to model clients [24]. In such a setup, the model can request an operation through a tool call, while the host application or server performs the actual operation. For a source-code analysis system, this separation is important. The analyzer performs detection and returns structured output, while the model handles conversational routing and explanation.

Tool use does not remove the grounding problem introduced in Section 2.4. A model may call the correct tool and still misread the result, ignore nested fields, or generalize beyond the returned data. The same issue appears in retrieval-augmented generation, because an answer can be conditioned on returned material and still fail to stay faithful to it.

DotArchSentry follows this tool-using and externally grounded pattern, but in a constrained form. The model is allowed to select analysis and reporting tools, and it can explain the packets returned by those tools. It is not used as the analyzer. The source-derived architectural facts come from Roslyn extraction, graph analysis, and packet construction. The LLM layer presents those facts conversationally and is checked against the packet boundary described later in Chapter 8.

3.3 Positioning of DotArchSentry

The surveyed tools define the space around DotArchSentry along two dimensions. The first is analysis scope. SonarQube represents broad project-quality governance, NDepend represents .NET dependency and metric inspection, and Arcan represents architecture-smell detection from dependency models. The second is how architectural intent

enters the tool. ArchUnitNET and NetArchTest rely on rules written by the developer, while smell-detection tools search the dependency model for structures that may deserve inspection.

DotArchSentry is closest to NDepend and Arcan on the detection side. Like them, it treats dependencies as structural data. Its analyzers report cycles, hub-like coupling, god-class candidates, and layer violations from a graph extracted from a C# solution. The implementation is narrower than those tools, but it follows the same general idea that architectural problems often appear as relations among several program elements.

For conformance checking, DotArchSentry is closer to architecture-test libraries. Its blueprint file states allowed layer dependencies and gives the layer analyzers a policy to check. DotArchSentry stores the policy as analysis configuration and evaluates it during an architectural run, so layer checking appears in the same report as cycles and hubs.

The delivery side is where DotArchSentry differs most from the surveyed architecture-analysis tools. The system is designed so that findings can be consumed through HTTP API workflows and through a local LLM conversation. The analyzer still produces the findings, while the delivery layer changes how users inspect, compare, and discuss them.

DotArchSentry's contribution is this particular combination: C# dependency extraction, selected architecture findings, persisted run comparison, packetized reporting, and local LLM-assisted presentation under grounding controls. The analyzer remains the source of architectural claims, while the LLM gives users a conversational way to inspect and discuss those claims. The central question is whether this delivery layer can make findings easier to work with without weakening the repeatability of the static analysis.

4 Solution Architecture

This chapter presents the overall architecture of DotArchSentry. It introduces the main user-facing workflows, the project-level components, and the persistent artifacts behind the solution. The following chapters then describe dependency extraction, architectural analysis, and grounded reporting in more detail.

4.1 Overview and Main Use Cases

DotArchSentry is intended for developers and software architects who need to inspect the dependency structure of a C# solution. From the user's perspective, the system is organized around a small set of workflows: analyzing a solution, configuring architectural policy, inspecting findings, requesting recommendations, and comparing runs.

The main use cases supported by the system are the following:

- running an architectural analysis on a C# solution using a selected analysis-depth preset,
- generating or loading an architecture blueprint,
- listing the findings produced by an analysis run,
- inspecting a finding to understand why it was detected,
- requesting refactoring recommendations for a finding,
- comparing two runs of the same solution.

After an analysis run exists, inspection and fix-option requests remain separate. This split matters later, in Chapter 8, because each workflow gives the LLM a different packet shape and a different grounding boundary.

The same workflows are exposed at three interface levels. The HTTP API exposes the core operations directly. The MCP server (Section 3.2) maps selected API workflows into tools that an LLM client can call. The OllamaBridge adds an interactive local command-line

interface over those tools. All three interfaces operate over the same persisted analysis and reporting model.

4.2 End-to-End Pipeline

The high-level pipeline of DotArchSentry starts with a C# solution and ends with structured outputs that can be consumed by users or LLM clients. Figure 4.1 summarizes the primary analysis-run path and the reporting workflows that consume the persisted result.

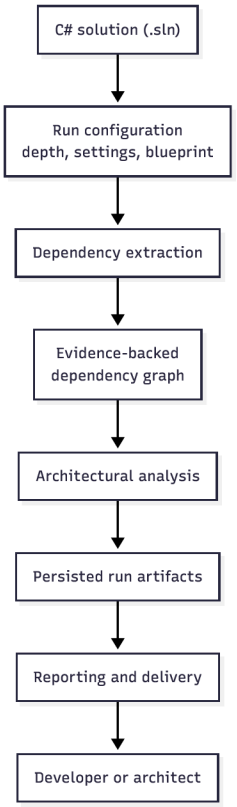

Figure 4.1: High-level DotArchSentry pipeline from solution input to reporting and delivery.

Before extraction, DotArchSentry resolves the run context: it validates the solution path, locates the solution root, prepares the local `.dotarchsentry` directory, and combines the selected depth with

saved per-solution policy. A saved blueprint also changes the run context, because it enables layer analysis. Roslyn and MSBuild then load the solution and produce the type-level dependency graph described in Chapter 5. The graph is passed to the architectural analyzers, which report cycles, hub-like coupling, god-class candidates, and layer violations. After scoring and enrichment, the API persists the run. Later workflows read different projections of that stored result: inspection and fix-options use the full report, while listing and comparison can use summaries and run context.

4.3 Subsystems and Architectural Boundaries

DotArchSentry is split into several projects, each responsible for a different architectural layer. This separation keeps source extraction, analysis, transport contracts, API orchestration, MCP access, and local LLM interaction in separate modules.

`DotArchSentry.Core` contains shared primitives used by the extraction and analysis layers. It defines the dependency graph model, node and edge records, dependency kinds, evidence structures, extraction options, project inclusion and exclusion information, and the per-solution settings schema. These types represent the source-derived structural model and the configuration that controls what is considered in scope.

`DotArchSentry.Roslyn` is responsible for extracting dependency information from C# solutions. It loads projects through Roslyn/MSBuild, collects declared types, extracts signature dependencies, extracts selected body-level dependencies, filters references based on extraction options, and tags ORM entities. This project is concerned with turning source code into graph data. It does not decide whether a dependency structure is architecturally problematic.

`DotArchSentry.Analysis` contains the architectural analysis layer and the reporting pipeline. The analyzers operate over the extraction outputs and produce finding artifacts. This project also contains logic for layer-rule resolution, finding construction, severity-related metrics, fact extraction, interpretation, strategy selection, and packet construction for inspection and recommendation workflows. The analysis project therefore covers two stages: detection (the analyzers in

Section 6.3) and the fact-extraction-plus-interpretation pipeline (Section 7.2) that turns each finding into a typed packet for inspection and recommendation.

`DotArchSentry.Contracts` defines the reporting vocabulary shared across analysis, API, MCP, and OllamaBridge layers: finding kinds, interpretation and strategy kinds, packet DTOs, kind-specific metrics, and shared JSON serialization conventions.

`DotArchSentry.Api` is the application layer of the system. It exposes HTTP endpoints for running analysis, managing settings, reading reports and findings, generating or retrieving blueprints, inspecting findings, generating fix options, and comparing runs. The API also owns API-specific request and response types, because these describe the HTTP interface rather than the internal analysis model. Internally, the API coordinates extraction, analysis, scoring, persistence, baseline handling, blueprint loading, and comparison services.

`DotArchSentry.Mcp` exposes selected API workflows as MCP tools. It is intentionally thin: each tool call is forwarded to the corresponding HTTP endpoint, and the API's JSON response is returned through the MCP response.

`DotArchSentry.OllamaBridge` provides the interactive local LLM interface. It connects the MCP tools with a local Ollama model, maintains conversation state, renders grounded briefs, and applies grounding controls. It does not perform analysis itself; it presents structured outputs conversationally.

The project boundaries are reinforced by a separation between the data models used at different stages of the pipeline:

```
Core graph model
  -> Analysis artifact model
  -> API and contract DTOs
  -> LLM-facing presentation briefs
```

Each transition reshapes the data: API DTOs omit fields the analyzer keeps internally, and the LLM-facing brief is a presentation projection whose text must never be parsed back into facts by downstream code.

Figure 4.2 summarizes the project-level boundary structure and distinguishes compile-time project references from runtime communication.

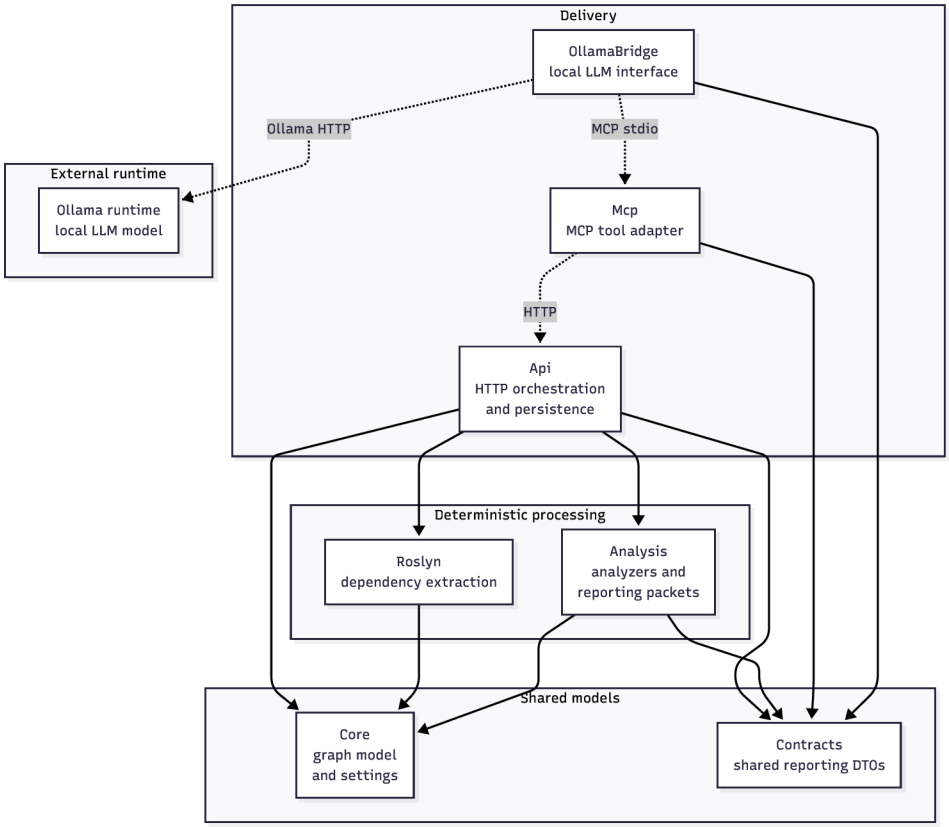

Figure 4.2: DotArchSentry project boundaries and dependency direction. Solid arrows denote project references; dashed arrows denote runtime communication.

4.4 Configuration, Blueprints, and Persistence

DotArchSentry stores per-solution artifacts under a `.dotarchsentry` directory next to the analyzed solution. These artifacts make the analysis repeatable, allow completed runs to be inspected later, and support comparison between runs. The persistence model is file-based JSON. SQLite was the obvious alternative for indexed querying, but does not fit this workload: each run produces a handful of artifacts (three findings in the example run, on the order of dozens for a real codebase), runs are written once and never updated, comparison reads two run directories without any cross-run join, and every bump of analysisHeuristicsVersion would require a schema migration. As a side benefit, JSON artifacts diff under git diff and can be inspected with jq while heuristics are being tuned.

Per-solution settings. The per-solution settings file is `dotarchsentry.json`. It represents architectural policy for the solution, such as user-defined project and namespace exclusions and analyzer sensitivity settings. Runtime extraction choices, such as the selected analysis depth, are not persisted as global policy in this file; they are resolved for each run and stored in the run context.

Analysis depth. The selected depth is not stored as permanent solution policy; it is resolved per run and recorded in the run context. Chapter 5 describes the depth presets.

Architecture blueprint. The architecture blueprint is stored as `architecture-blueprint.json`. A blueprint contains layer definitions and dependency rules. When a saved blueprint exists, the analysis run can check dependencies against these rules and report layer violations. DotArchSentry can generate a project-scope or namespace-scope blueprint from the currently observed structure. The generated file captures the architecture that the code currently exhibits, including dependencies that an architect would want to forbid, and must be reviewed before being used as policy.

Baselines. The baseline file, `baseline.json`, records findings that have been accepted or intentionally ignored by the user. Baselines are display tags. The analyzers still record baselined findings, and the persisted run still contains them. This is what lets run comparison report whether a finding was baselined in the reference run, in the candidate run, or in both.

Per-run artifacts. Each analysis run is stored under `.dotarchsentry/runs/<run-id>/`. A run contains:

```
metadata.json
report.full.json
findings.summary.json
run-context.json
```

The run metadata records the run identifier, when the analysis was created, which solution was analyzed, how many findings were produced, and which analysis-heuristic version was used. The full report stores the complete enriched finding set for the run. The findings summary is a smaller projection of scored findings used when the full evidence payload is not needed, especially for listing and comparison. The run context stores the settings and scope under which the run was produced, including the resolved extraction settings, analysis depth, included and excluded scope, blueprint drift information, and baseline snapshot.

The split is load-aware. Inspection and fix-options read the full report (Section 7.3) for the source-level evidence they cite. Comparison reads only the summary and run context (Section 7.4), since it operates on metrics and identities rather than evidence.

5 Dependency Extraction and Graph Model

This chapter describes how DotArchSentry extracts architectural dependency information from C# solutions and represents it as a dependency graph. In this thesis, a dependency graph is a directed graph whose nodes represent program elements, primarily C# types, and whose edges represent typed dependency relationships between them. The extraction layer is the first stage of the system: it turns source code into this structural model.

5.1 Input Scope and Extraction Context

DotArchSentry analyzes C# systems from a `.sln` file rather than from an individual project or source-code directory. Solution loading, scope resolution, and dependency extraction are handled by the `DotArchSentry.Roslyn` component. This solution-level input matches the architectural level targeted by the tool, because relevant dependencies often cross project boundaries and project references may themselves express architectural structure.

Before construction, the runtime extraction options are merged with the per-solution policy file, which can contain project exclusions, namespace exclusions, and an architecture blueprint. The merged result is the extraction context for the run.

Roslyn exposes symbols from many sources beyond the analyzed projects themselves, including NuGet packages, framework assemblies, generated code, and test projects. DotArchSentry uses depth-controlled extraction settings to decide which of these become graph nodes and dependency edges:

- `Internal`: includes solution-internal code and excludes external types, system namespaces, generated code, and test projects.
- `Extended`: includes external types, such as types from NuGet dependencies, but still excludes system namespaces, generated code, and test projects.
- `Full`: includes external types, system namespaces, generated code, and test projects.

These presets affect graph construction, not just the final report. Depending on the selected depth, types in excluded scopes are not added as graph nodes, and references targeting excluded types are not added as dependency edges. The underlying solution may still be loaded by Roslyn, using the mechanisms introduced in Section 2.3, but the resulting graph represents only the scope selected for the run. In practice, `Internal` is the normal mode for architectural analysis, `Extended` is useful when library coupling is part of the question, and `Full` is mainly an exploratory mode because it can substantially increase graph size, persistence size, analyzer cost, and interpretive noise.

Depth presets can be further refined using explicit project and namespace exclusions. Project exclusions remove whole projects from the graph. Namespace exclusions apply both to source types and dependency targets, so excluded namespaces do not appear as graph nodes or edge targets. DotArchSentry applies these filters during extraction, because they affect both analysis cost and analysis meaning. The resolved extraction context is persisted with each run. This allows findings, explanations, and the run comparison workflow described in Chapter 7 to be interpreted against the scope that produced them.

5.2 Graph Construction Pipeline

DotArchSentry turns a C# solution into the type-level dependency graph used by later analyzers through a load-and-resolve pipeline. The pipeline does not yet decide whether a dependency is architecturally interesting. That decision belongs to the analyzers in Chapter 6. The work is coordinated by `GraphBuilder` and proceeds in five stages, shown in Figure 5.1: load the solution, select the C# projects in scope, build an assembly-to-project index, collect declared types, and invoke the dependency extractors.

The solution is opened through Roslyn's `MSBuildWorkspace`. What Roslyn can resolve depends on more than source files. Project references, target frameworks, and compiler options all influence symbol resolution. If the workspace cannot open the solution, the run aborts. Partial symbol resolution, where the solution opens but some references stay unresolved, is treated as an extraction limitation (Sec-

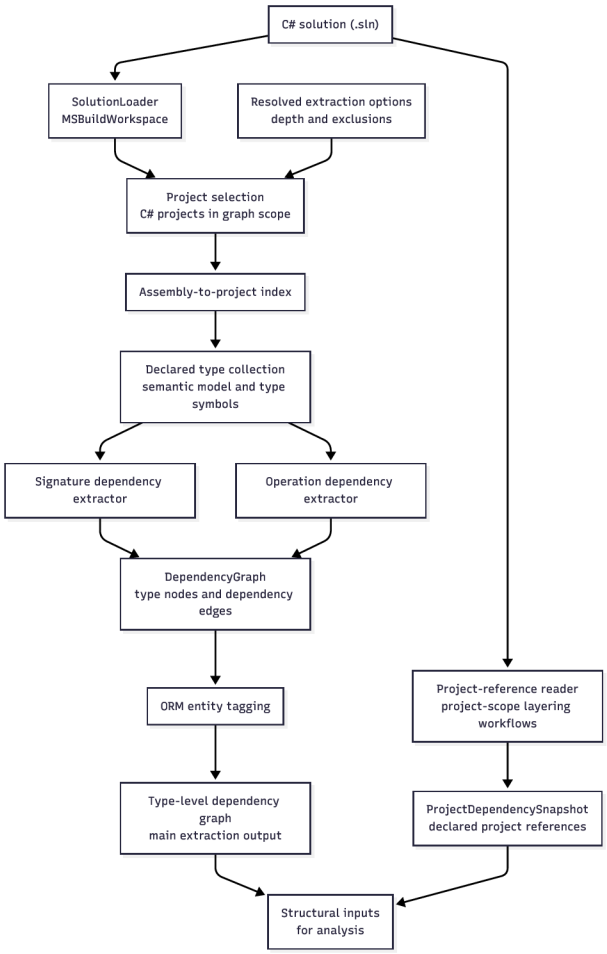

Figure 5.1: Dependency extraction paths. The type-level dependency graph is the main extraction output. A separate project-dependency snapshot records declared project references for layer analysis at project scope.

tion 5.6). After loading, `GraphBuilder` selects participating C# projects and applies user-defined project exclusions.

The assembly-to-project index built next separates solution-internal symbols from NuGet or framework symbols. This distinction governs filtering and node creation. Internal dependency targets can become graph nodes at `Internal` depth, while external targets are admitted only when the active depth allows them. For each remaining project, `GraphBuilder` obtains a Roslyn `Compilation` and resolves type and delegate declarations to Roslyn `INamedTypeSymbol` instances. Top-level statements are handled through Roslyn's generated program type when it is exposed. Each resolved declaration (`DeclaredType`) carries its syntax node, semantic model, and compilation context forward, so the extractors operate on resolved symbols rather than on names.

The final stage invokes `SignatureDependencyExtractor` and `OperationDependencyExtractor`. Section 5.4 catalogs the dependency kinds they emit.

5.3 Extraction Outputs: Type Graph and Project Snapshot

The extraction layer produces two structural outputs from the analyzed solution. The first is the type-level dependency graph, represented by `DependencyGraph`. The second is the project-dependency snapshot, represented by `ProjectDependencySnapshot`. Both are derived from the same solution-level input, but they capture different architectural facts.

The type-level graph records resolved dependencies between C# types. Its nodes represent declared types such as classes, interfaces, structs, records, enums, and delegates. Its edges represent typed dependencies between those nodes, with the concrete dependency kinds defined in Section 5.4. This graph is the main input for analyses that reason about type-level structure, including cycle detection, hub detection, namespace-scope layer violations, and evidence-backed reporting.

The project-dependency snapshot records declared project-to-project references from the solution structure. This is intentionally

separate from the type-level graph because a declared project reference is not equivalent to an observed type-level dependency. A project may reference another project even if no extracted type edge currently uses it, for example because the reference is unused or only exercised through code paths excluded from extraction. The type graph captures resolved type usage but does not model projects as ordinary graph nodes. The dependency kind `ProjectReference` marks edges that come from this snapshot rather than from the type-level extractors.

5.4 Signature and Body-Level Dependency Extraction

DotArchSentry extracts type-level dependencies using two complementary strategies. Signature-level extraction inspects declarations and member signatures. Body-level extraction inspects supported executable code inside members and initializers. Both strategies operate on the resolved `DeclaredType` objects prepared by the graph construction pipeline, and both add edges to the same type-level dependency graph.

Signature-level extraction captures dependencies that are visible from declarations and member signatures, without inspecting executable method bodies. These dependencies are treated as structural because they describe how a type is declared, exposed, or constrained.

The signature extractor records the following dependency kinds:

- `Inheritance`: a type inherits from a base type.
- `InterfaceImplementation`: a type implements an interface.
- `FieldType`, `PropertyType`, and `EventType`: a member stores or exposes another type.
- `MethodReturnType` and `MethodParameterType`: a method signature exposes another type.
- `GenericConstraint`: a generic type parameter is constrained by another type.
- `Attribute`: a type or member is annotated with an attribute type.

For compound type references, the extractor expands the referenced type symbol to named types before creating graph edges. This means that dependencies inside generic type arguments, arrays, pointers, and tuple elements are represented explicitly in the graph. For example, a member of type `IRepository<Customer>` contributes dependencies not only to the repository abstraction but also to `Customer`.

Body-level extraction captures dependencies that arise from implementation logic rather than from declarations. This matters because a type can expose a simple signature while still creating concrete collaborators, invoking services, accessing static members, or catching framework-specific exceptions inside method bodies.

The operation extractor operates on Roslyn's operation model rather than treating every syntax node as a dependency source. It obtains operation roots for explicitly supported executable constructs, including method and constructor bodies, destructors, accessors, property, field, and event initializers, expression-bodied members, local functions, operators, conversion operators, and global statements in top-level programs when Roslyn can produce operations for them. From these operation trees, it records a fixed subset of operation kinds that expose type-level dependencies useful for architectural analysis:

- `ObjectCreation`: an object is instantiated.
- `Invocation`: a method call targets a type.
- `MemberAccess`: a field, property, or other member is accessed.
- `TypeOf`: a `typeof` expression references a type.
- `CastOrConversion`: a cast or conversion targets a type.
- `PatternMatch`: an `is`-type expression (`x is T`) checks against a type.
- `CatchType`: an exception handler catches a specific exception type.

Keeping both allows the graph to distinguish a stable structural relationship, such as a field type or method parameter, from a behavioral relationship, such as a method invocation or object creation.

5.5 Evidence, Enrichment, and Graph Representation

A dependency edge is not only a relationship between two graph nodes. To support later analysis and explanation, DotArchSentry stores typed evidence on edges and structural counters on nodes, and tags nodes that represent persisted domain entities.

The graph is represented by `DependencyGraph`, which owns a set of `Node` records and a collection of `Edge` records. Node kinds correspond to extracted C# type categories: classes, interfaces, structs, enums, delegates, record classes, and record structs. Each node also stores:

- identification: the symbol identifier, display name, namespace, project, and assembly,
- structural counters (method, property, and field counts) precomputed during extraction so analyses do not re-inspect declarations.

An edge carries the source and target node identifiers, the dependency kind from Section 5.4, an occurrence count, and a list of evidence entries. Internally, the graph deduplicates edges by the `(from, to, kind)` triple: when the same triple is observed again, the count is incremented and the new evidence entry is appended to the existing edge. For example, three different methods of `OrderService` that all call `EmailGateway.Send` are recorded as a single `Invocation` edge with count three and three evidence entries, rather than as three separate edges.

Each evidence entry is a self-contained record of a single observed location. It stores the source file path, the one-based line number, a short source snippet, and the enclosing member where the dependency was observed when applicable. The enclosing member is recorded as a compact qualified label, for example `OrderService.PlaceOrder` for a method call. This lets later reporting point to the exact program element that introduced a dependency, instead of only to the type that declares it.

When a dependency kind needs more than a source location to be interpreted, an evidence entry also carries typed metadata, a structured record which can include:

- the name of the invoked method, for `Invocation` edges,
- the name of the accessed member, for `MemberAccess` edges,
- the C# visibility of the source member that introduced the dependency, for signature-level edges,
- for object creation, a classification of the construction site (constructor body, initializer, accessor, etc.) and of how the new instance is stored.

These facts are not used by the extractor itself. They are persisted with the edge evidence so that the reporting pipeline described in Chapter 7 can extract structured facts from a finding without rewalking syntax. The cost of syntactic and semantic inspection is therefore paid once during extraction and reused by every later interpretation of the run.

Both extractors emit evidence in this same shape, but attach it at different syntactic levels. Signature-level evidence is attached to the declaration or member that introduces the dependency, such as a field, property, or method signature. Operation-level evidence is attached to the operation syntax node that caused the dependency, which is also where the operation-specific metadata described above is added. This keeps the evidence model uniform while still allowing body-level dependencies to carry richer context.

After all extractors have run, DotArchSentry performs one node-level enrichment pass that tags graph nodes representing persisted domain entities. The tagging seeds from Entity Framework Core `DbContext` subclasses, collects the type arguments of their `DbSet<T>` properties [25], and propagates the tag transitively through fields and properties of already-tagged types, including type arguments of common collection types. The tag is consumed later by analyses and reporting, where the distinction between an ORM entity and an ordinary class affects how a coupling pattern should be interpreted. The detection is heuristic and structural rather than attribute-based; its limitations are revisited in Section 5.6.

5.6 Extraction Limitations

The dependency graph is a bounded representation of the analyzed solution. Its boundaries come from the Roslyn and MSBuild loading pipeline, from the limits of static analysis, and from DotArchSentry's own choices about scope, extraction strategy, and heuristic enrichment. None of these limits invalidates the dependency graph, but they shape how its findings should be interpreted.

Solution loading. A first distinction concerns how Roslyn loads a solution. If `MSBuildWorkspace` cannot open the solution at all, for example because a required SDK or a referenced project is missing, the run is aborted as a hard load failure and no graph is produced. A more subtle case is partial resolution, where the workspace opens but Roslyn cannot fully resolve every symbol referenced from the loaded projects. This can happen when target frameworks are mismatched, when package restoration is incomplete, or when project references point to projects that the workspace declined to load.

In this case, extraction proceeds and produces a dependency graph, but some references in that graph may be missing because the underlying symbols were not resolvable. Findings produced from such a run are valid only relative to the resolvable subset. This is the more important failure mode for evaluation purposes, because it can leave analyses silently incomplete rather than failing loudly.

Static-analysis blind spots. Beyond resolution problems, some limits are inherent to static analysis. Even when Roslyn resolves every reachable symbol, several relationships in a real C# program are not reliably recoverable from static type references alone:

- reflection-based access to types and members, including dynamically loaded assemblies,
- runtime dependency injection, where the concrete type bound to an interface is determined by a container at startup rather than by a constructor call,
- runtime configuration that selects an implementation through a string identifier or factory delegate,
- dynamic dispatch via the `dynamic` keyword, which defers member resolution to the runtime.

Such relationships can appear in the source as a reference to an abstraction with no statically resolvable provider. The graph therefore reflects observed type usage and not the actual runtime topology. Architectural conclusions drawn from the graph apply to the statically visible structure of the system.

Scope and exclusions. The depth presets and explicit exclusions described in Section 5.1 introduce a deliberate but real limitation. Anything excluded from the graph is invisible to subsequent analyses, so a coupling pattern that runs through an excluded namespace, project, or generated document does not appear in any finding at the chosen depth. One could argue that `Extended` should also include test projects, since tests are part of the solution and often reveal how production components are exercised. DotArchSentry keeps them excluded at this depth because test code usually follows a different dependency structure from production code: it may intentionally cross architectural boundaries, assemble many components in fixtures, or depend on infrastructure used only for verification.

Extraction strategy tradeoffs. The two extraction strategies can be disabled for a run, but this changes the meaning of the graph. Disabling body extraction removes `Invocation`, `MemberAccess`, and `ObjectCreation` edges, weakening analyses that depend on implementation behavior. Disabling signature extraction removes declared relationships such as `Inheritance`, `FieldType`, and `MethodParameterType` dependencies, which can make cycles, hubs, and layer violations appear weaker than they are.

Heuristic enrichment. The ORM entity tagging described in Section 5.5 is also limited by its own design. The detector seeds from Entity Framework Core's `DbContext` and reaches entities through navigation properties and collection types [25, 26]. Entities defined through other persistence frameworks, through configuration files or fluent mapping APIs, or through attribute-based mapping with no `DbContext` surface are not tagged. The detection is therefore a useful heuristic for EF Core codebases, but absence of the tag should not be read as evidence that a type is not a persisted entity.

6 Architectural Analysis and Finding Artifacts

The analysis layer decides whether a dependency structure is architecturally problematic. It consumes the dependency graph produced in Chapter 5 and produces internal finding artifacts that capture cycles, hub-like coupling, god-class indicators, and layer violations.

Each finding is a stable, evidence-backed record. It carries the graph evidence and structured measurements on which severity is later computed. These findings are the input to the reporting pipeline of Chapter 7 and the LLM-presented explanations of Chapter 8. Here, the artifacts are produced, and later they are presented.

6.1 Analyzer Pipeline and Sensitivity Settings

The analysis layer does not consume the raw dependency graph directly. Before traversal-heavy analyses run, the graph is projected into a traversal-oriented index that groups outgoing and incoming edges by node and exposes the lookups the analyzers need. The index, called `GraphIndex`, exposes nodes by identifier, an enumeration of all nodes, the included and excluded project metadata, and per-node access to outgoing and incoming edges. These lookups support constant-time node access and direct adjacency traversal, which would otherwise require linear scans over the unindexed edge collection.

During index construction, edges whose source and target are the same type are excluded from the traversal index. The raw dependency graph may contain such self-dependencies, but the analyzers operate on cross-type relationships, so self-loops are not exposed through the analysis index.

Detection is performed by a small fixed analyzer family. The cycle and hub analyzers run for every analysis. Layer analysis is added only when a saved blueprint is available. The blueprint scope determines whether the namespace-scope or project-scope layer-violation analyzer is instantiated. The namespace-scope analyzer operates over type-level graph edges, while the project-scope analyzer evaluates declared project references from the project-dependency snapshot. The set is fixed today, but the analyzers follow the same output contract. Any future analyzer would be expected to consume a resolved structural

input and emit the same finding artifact shape. The application collects all findings from the active analyzers into one list, then builds the full report with graph-neighborhood enrichment and scores the findings before persistence, as summarized in Figure 6.1.

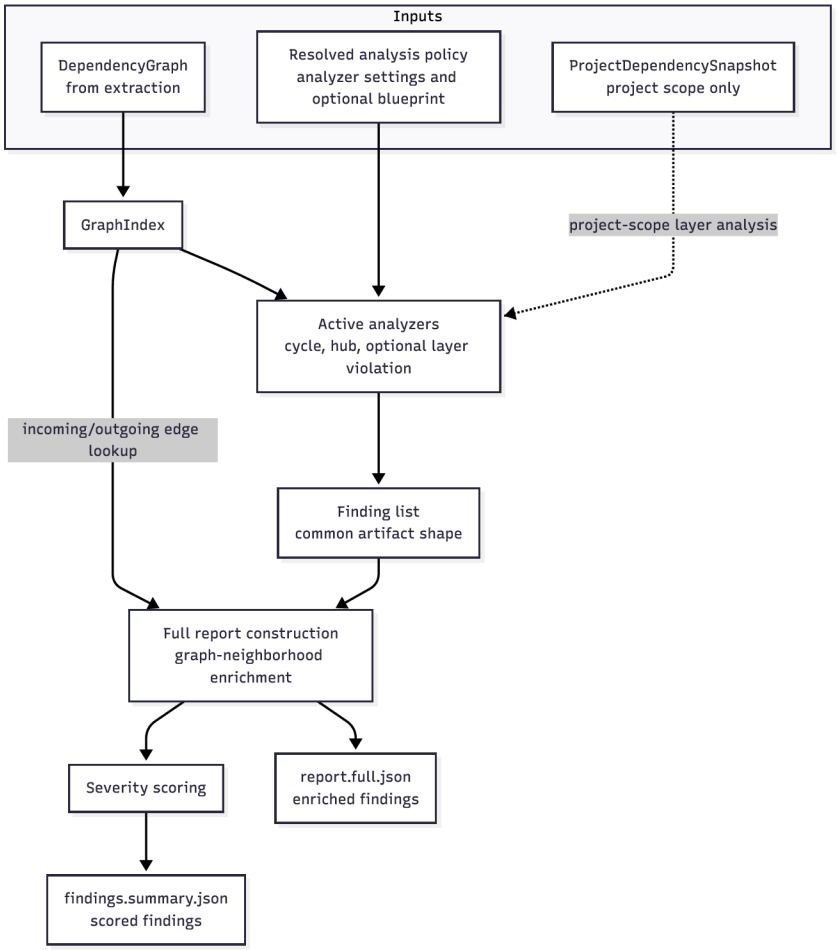

Figure 6.1: Analyzer pipeline from analysis inputs to enriched and scored finding artifacts.

The behavior of the analyzers is controlled by a per-solution policy resolved before each run. The policy supplies per-analyzer sensitivity thresholds and reporting caps. When no policy file exists for a solution, a default policy is created with values chosen to produce

a manageable amount of output on typical mid-sized C# codebases. The most important settings are the following.

Hub thresholds. The hub analyzer combines absolute, relative, and percentile-based thresholds. Absolute thresholds prevent very small fan counts from being reported as hubs. Relative thresholds scale the decision with graph size. Percentile thresholds then require a candidate to be prominent compared with the rest of the graph, either by raw fan or by strength-weighted dependency score. A separate floor on non-trivial method count is used when distinguishing god-class candidates from interface or data-like hubs. The reporting cap limits output volume, but it should be read as a noise-control mechanism rather than as a claim that unreported candidates are architecturally irrelevant.

Cycle suppression. The cycle analyzer is mainly controlled through suppression rather than thresholds, because the underlying detection is exact. Cycles in which every member is tagged as an ORM entity are discarded when the corresponding setting is enabled, because cycles like these are typically a consequence of bidirectional navigation properties between persisted entities and do not indicate an architectural problem. Cycles whose members all fall under a configured suppressed-namespace prefix are also discarded. A hard cap limits the number of cycles reported.

Layer-violation cap. The layer analyzers expose only a cap on the number of reported violations. The set of violations is otherwise determined entirely by the blueprint described in Section 6.2.

6.2 Architecture Blueprints

A blueprint is the architectural policy against which layer analyzers operate. Without a blueprint, the layer analyzers have no notion of which dependencies are allowed and therefore produce no findings. A blueprint is therefore an input to analysis rather than a result of it.

The blueprint model is small. A `LayerRuleSet` consists of layer definitions, dependency rules, and resolver flags. A `LayerDefinition` names a layer and supplies the patterns by which nodes are mapped to it. A definition can be expressed as a list of project wildcard patterns, as a list of namespace prefixes, or both. A `LayerDependencyRule`

expresses an allowed dependency: it states that nodes mapped to the source layer are permitted to depend on nodes mapped to any of the listed target layers. Dependencies not covered by an explicit rule are forbidden by default, unless the rule set is configured to allow unspecified dependencies. The resolver flags determine whether dependencies within the same layer are allowed and how unmapped nodes are treated.

DotArchSentry can produce two kinds of blueprint, distinguished by the scope at which architectural layering is expressed.

Project-scope blueprint. A project-scope blueprint treats each project in the solution as a layer. The generator reads the project-dependency snapshot introduced in Section 5.3, creates one `LayerDefinition` per project name, and infers two-segment namespace prefixes for each definition based on observed namespaces. The dependency rules are then derived from the snapshot itself: for each project, the generator emits one rule allowing dependencies to the projects that the project actually references. The resulting rule set encodes the currently observed project graph as the architectural policy. This is appropriate for solutions where the project structure is the main carrier of architectural intent.

Namespace-scope blueprint. A namespace-scope blueprint is finer-grained. The generator reads distinct project-namespace pairs from the type-level graph, creates one `LayerDefinition` per pair, and derives rules from the observed type-level edges between layers. A type-level edge whose source and target fall in different layers contributes one allowed cross-layer dependency. Namespace-scope rules therefore reflect not only which projects communicate, but which namespaces inside those projects reach across architectural lines.

A generated blueprint is an editable starting point, not a verdict that the current architecture is correct. It captures the architecture that the code currently exhibits, which may include the very dependencies that an architect would want to forbid. The expectation is that a generated blueprint is reviewed and edited before being treated as policy.

The workflow therefore involves blueprint generation followed by at least one analysis run. The generator produces a baseline blueprint from the observed structure; running layer analysis against that unmodified baseline yields no findings, because the rules cover every existing dependency by construction. The architect then edits the

blueprint to encode intent (typically by removing one or more allowed dependencies, or by adding a new layer that has no incoming rules) or changes the code to introduce a dependency the rules do not permit. The next analysis run reports any dependency that crosses the resulting policy as a layer-violation finding.

Persistence and drift. A blueprint is stored as `architecture-blueprint.json` in the per-solution `.dotarchsentry` directory described in Section 4.4. When a saved blueprint is present at run time, the corresponding layer analyzer is enabled automatically. The analysis service additionally compares the saved blueprint against the current graph and records drift items when currently observed projects or namespaces are not covered by any layer in the saved blueprint. Drift detection is informational: it does not modify the saved blueprint and does not gate the analysis. It is used by run comparison and reporting to flag situations where a blueprint may be out of date.

6.3 Detected Architectural Issues

These issues are produced by four analyzer types. A cycle analyzer, a hub analyzer, and two layer-violation analyzers operating at namespace and project scope, of which at most three run in any single analysis (the two layer analyzers are mutually exclusive, as Section 6.1 describes). Each analyzer consumes its structural input from the extraction layer and, where applicable, a blueprint. The findings share a common artifact shape. The differences lie in the detection logic and measurements produced by each analyzer.

Cycles. The `CycleAnalyzer` detects strongly connected components of size greater than one in the type-level dependency graph. The detection algorithm follows the Kosaraju–Sharir two-pass linear-time algorithm for strongly connected components [2], implemented in `KosarajuScc` over the index's outgoing and incoming adjacency. Each non-trivial SCC is treated as a candidate cycle. Two suppression heuristics are then applied: cycles in which every member is tagged as an ORM entity are discarded when the corresponding setting is enabled, and cycles whose members all fall under one of the configured suppressed-namespace prefixes are also discarded. The remaining cycles are reported with a `CycleMetrics` record that captures

the number of members, the number of internal edges, the structural and behavioral edge strength, the project and namespace spread, and a flag indicating whether any internal edge is an inheritance edge.

Hub-like coupling. Hub-like dependency is a known architectural smell in dependency-graph-based architecture analysis tools such as Arcan [20]. DotArchSentry uses the same general intuition: a type becomes architecturally interesting when it has disproportionate incoming or outgoing connectivity relative to the rest of the graph. Before evaluating hub candidates, the `HubAnalyzer` skips nodes that look like passive data carriers, such as DTOs, request or response models, or simple entities. Such nodes can be widely referenced without being architecturally interesting hubs.

For each remaining candidate, the analyzer computes adjusted fan and adjusted strength scores. Passive data-like collaborators are not removed from these computations, but they contribute with reduced weight, so dependencies on data carriers remain visible without dominating hub eligibility. A node qualifies as a hub when its adjusted fan exceeds the relative or absolute floor and either its adjusted fan or its adjusted strength score sits in the configured top percentile. Qualifying nodes are then passed to the hub classifier (`HubClassifier`), which assigns one of three sub-kinds.

Hub sub-kinds. The sub-kind assignment is specific to DotArchSentry. It separates hub candidates whose graph metrics have the same shape but whose architectural interpretation differs. A hub identified primarily by incoming dependencies is reported as `HubInterfaceCentrality` when the underlying type is an interface. This case is separated because high fan-in on an abstraction can be an intended design point, consistent with dependency-inversion style designs in which clients depend on abstractions rather than concrete implementations [27]. A concrete hub with broad outgoing coupling is reported as `GodClass` when it also has a larger behavioral surface, multiple stored collaborators, diffuse internal workflows, and a broad rather than concentrated outgoing surface. This adapts established metric-based God Class detection strategies, which treat large, responsibility-heavy classes with extensive external collaboration as design flaws [3, 4]. Remaining concrete hubs are reported as `HubImplementationHotspot`: they are central coupling points, but they do not satisfy the stronger conditions used for the god-class sub-kind. The three sub-kinds share

the same `HubMetrics` shape and are distinguished at the finding level by their `FindingKind`.

Layer violations. Layer violations are reported by two analyzers that differ in scope. The `NamespaceScopeLayerViolationAnalyzer` iterates over type-level edges in the graph, resolves each source and target to a layer through a namespace-aware resolver, and reports an edge as a violation when its cross-layer dependency is not permitted by the rule set. Edges are grouped by source-target type pair so that multiple dependency kinds between the same pair collapse into a single finding. The `ProjectScopeLayerViolationAnalyzer` performs the same check at project granularity, using the project-dependency snapshot rather than the type graph, and produces one finding per violating project reference. Both analyzers emit `LayerViolationMetrics` that record the number of crossings, a flag indicating which scope produced the finding, and the structural and behavioral edge strength.

6.4 Finding Model, Metrics, and Severity

A finding is the unit of analysis output. It is an internal artifact rather than a transport object: it is produced by an analyzer, enriched before persistence, and paired with a severity score in the persisted summary. It is projected to API or LLM-facing forms only when needed.

Finding shape. Each `Finding` record carries a stable identifier, the finding kind, a short title, an aggregated metrics object, the list of involved member references, the list of evidence edges drawn from the graph, and the node neighborhoods added during enrichment. Two optional fields capture finding-specific information: a `LayerBoundary` record present on layer-violation findings, and a `KindMetrics` record.

Polymorphic kind metrics. The kind-specific metric records are unified under a polymorphic `KindMetrics` type whose JSON discriminator is named `metricFamily`. The concrete metric records are `CycleMetrics`, `HubMetrics`, and `LayerViolationMetrics`. They are read by the severity scorer, by the fact-extraction stage of the reporting pipeline in Chapter 7, and by the run-comparison workflow. The metric record is the canonical source of per-finding measurements for these consumers.

Member references and evidence edges. A `NodeReference` is a snapshot of a graph node at the moment the finding was produced. It carries identification fields and the same structural counters that the `Node` record carries in the graph. A `FindingEdge` is the corresponding snapshot of a graph edge. It stores the source and target identifiers, the dependency kind, the occurrence count, and the evidence entries described in Section 5.5.

Synthetic node kinds. Finding artifacts need to represent both type-level and project-level members. Real extracted types keep their C# kind, but a project-scope layer violation refers to projects rather than to types. The artifact node-kind enumeration therefore includes one synthetic value, `Project`, which does not appear in the core node-kind enumeration used by extraction. This lets findings refer uniformly to type-scope and project-scope members through the same `NodeReference` record.

Dependency strength. Severity scoring and several analyzer heuristics need a single numeric value for an edge's architectural significance. DotArchSentry derives that value from a fixed mapping in `DependencyStrength` that assigns each `DepKind` a strength between zero and one. This follows object-oriented coupling literature that treats coupling as a family of relationships rather than a single uniform edge type [5]. The concrete weights are DotArchSentry's heuristic policy: declared structural dependencies, such as inheritance, interface implementation, generic constraints, field types, and project references, receive higher weights than behavioral dependencies such as invocations, member accesses, casts, and pattern matches. The same mapping is also used to rank and order evidence edges.

Severity formula. The severity scorer combines three terms with fixed weights:

- *coupling*, which captures the structural and behavioral strength of the relationships involved in the finding,
- *breadth*, which captures the size and spread of the affected portion of the graph,
- *ruleImpact*, a finding-kind-dependent term that encodes DotArchSentry's built-in severity policy.

The three terms are clamped to the unit interval and combined as a fixed weighted average. The weighting gives coupling the largest influence because the tool is primarily a dependency-structure analyzer. Breadth and rule impact then adjust the score for spread and finding-family-specific architectural importance:

$$\text{severity} = 0.50 \cdot \text{coupling} + 0.25 \cdot \text{breadth} + 0.25 \cdot \text{ruleImpact}.$$

These weights are a deterministic severity policy. Their purpose is to produce stable ordering among findings from the same run and comparable severity values between compatible runs.

The terms are computed from `KindMetrics` fields. Continuous metrics such as edge strength or member count are normalized through a saturating logarithm, so that very large graphs do not dominate the score purely by size.

The meaning of the three terms depends on the finding family. For cycles, coupling is derived from structural and behavioral edge strength, breadth combines member count with project and namespace spread, and rule impact is higher when the cycle includes inheritance. For hubs, coupling uses the stronger of the incoming and outgoing dependency sides, breadth combines fan with non-trivial method count, and rule impact depends on the hub sub-kind. For layer violations, namespace-scope coupling is derived from the violating edge strength, while project-scope findings treat a forbidden project reference as fully impactful.

The final severity is rounded to four decimal places when it is written. This is the value persisted in the findings summary and reused by the run-comparison workflow described in Chapter 7. Rounding at write time keeps the severity stable across the full report, the compacted report, and downstream projections.

Post-analysis enrichment. Finding enrichment runs after analyzer execution while the graph index is still available. The `FindingNeighborhoodEnricher` records, for each node referenced by a finding, the top incoming and outgoing neighbors ranked by dependency-strength points, together with the dominant incoming and outgoing dependency kinds. This neighborhood snapshot is attached to the finding so that later interpretation can describe local graph context.

Compaction. The `FindingCompactor` builds a smaller presentation-time projection of a finding. It limits the number of evidence entries

and, for non-cycle findings, samples the most architecturally significant edges. Compaction does not replace the canonical run artifact: the full enriched finding is persisted, and compacted forms are built for clients that do not need the full evidence payload.

Pipeline ordering. The analysis service applies these steps in a fixed order: extract the graph, build the index, run the analyzers, enrich the findings while the index is still available, and score the findings. The full enriched report and the scored findings summary are then persisted as separate run artifacts.

7 Reporting Pipeline

After analysis has produced findings, the reporting layer reads stored finding artifacts, run context, and source evidence, then projects them into typed structures for listing, inspection, recommendation, and comparison. The LLM integration in Chapter 8 consumes these structures, but packet construction happens before any language model is involved.

7.1 Consumers of Analysis Output

The same analysis run is read in different ways depending on the workflow. A listing workflow needs compact scored findings that can be filtered and sorted. A finding-inspection workflow needs enough detail to explain why one finding was reported. A recommendation workflow needs the same explanatory structure plus a bounded set of remediation strategies. A comparison workflow needs two run summaries and their contexts, not the full evidence payload.

The three persisted artifacts introduced in Section 4.4 separate these concerns. `findings.summary.json` carries identity and metrics for listing and comparison. `report.full.json` carries enriched evidence for inspection and recommendation. `run-context.json` carries the scope and policy that the comparison gate reads.

Baselines are treated as reporting metadata rather than analysis filters. They can hide findings from listing views, but they do not remove findings from persisted run artifacts or from run comparison.

7.2 Finding-to-Packet Transformation

Inspection and recommendation packets are built from a single stored finding. The packet builders first construct any family-specific context needed for that finding: cycle topology for cycles, hub direction and representative caller or collaborator groups for hubs, and boundary context for layer violations. They then run the shared packet path: fact extraction, interpretation, and packet-body assembly. Recommenda-

tion packets insert strategy selection between interpretation and body assembly.

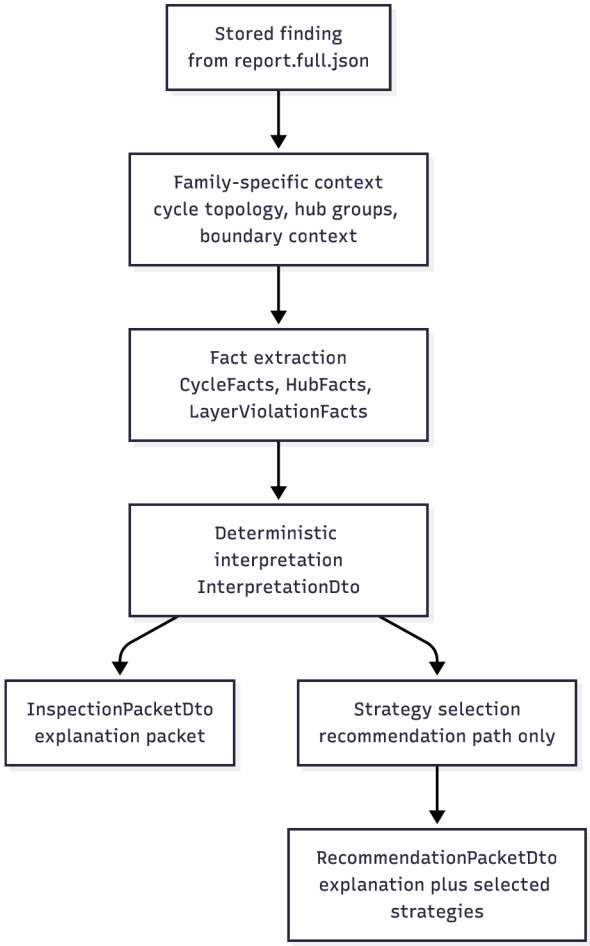

Figure 7.1: Finding-to-packet transformation for inspection and recommendation workflows.

Fact extraction. A fact extractor converts a finding into a typed `FindingFacts` record. The records are family-specific: `CycleFacts`, `HubFacts`, and `LayerViolationFacts`. They are structured measurements and boolean flags derived from the finding evidence and metrics. For example, cycle facts record whether the cycle contains object creation, field coupling, inheritance, redundant peer fields, or only

behavioral edges. Hub facts record fan direction, construction pressure, structural outgoing dependencies, and operational concentration. Layer-violation facts record the violated boundary, the crossing kinds, stored references, construction sites, signature crossings, and whether the boundary crossing is diffuse.

Interpretation. An interpreter maps facts to one or more `InterpretationDto` records, each with a fixed `InterpretationKind` and description. The kind names the architectural shape of the finding from a typed set rather than from free-form text. For example, an incoming concrete hub can be interpreted as a widely depended-upon implementation. The interpretation is related to Fowler's shotgun-surgery smell because changes to such an implementation can affect many direct callers [28]. A layer violation can be interpreted as boundary erosion, reflecting the idea that repeated architectural-rule violations contribute to architectural erosion [1]. The consumer always receives a known interpretation kind, and some interpreters mark one interpretation as dominant when several patterns apply.

Strategy selection. Strategy selection is used only for recommendation packets. The selector reads the finding and its interpretations and emits `StrategyDto` records drawn from a fixed set of `StrategyKind` values, such as `CentralizeObjectCreation`, `ExtractInterface`, `InvertDependency`, `NarrowInterface`, `RemoveProjectReference`, or `SplitResponsibility`. Candidate strategies are deduplicated by strategy kind and target edge, and a per-interpretation budget prevents one interpretation from dominating the result. The packet also records `AllowedStrategyKinds`, the exact set of strategy kinds selected for this finding. Chapter 8 uses this field as a grounding boundary for LLM-generated recommendations.

7.3 Inspection and Recommendation Packets

The inspection and recommendation workflows use the same packet-construction path. Both start from one stored finding and return a bounded projection of that finding. The separation exists because the two workflows answer different questions: inspection explains why a finding was reported, while recommendation adds possible remediation strategies.

An `InspectionPacketDto` contains the finding identity, metrics, interpretations, and a typed family-specific body. A `RecommendationPacketDto` keeps the same explanatory base and adds selected `StrategyDto` records together with the `AllowedStrategyKinds` set.

The packet body contains the information that is specific to each finding family:

- Cycle packets classify the cycle as `CycleMutualPair`, `CycleLinearChain`, or `CycleEntangledMesh`. They include the cycle members, shape, internal edge density, declared structural, behavioral, and construction counts, representative closed paths, cut candidates, grouped behavioral evidence, and representative evidence sites.
- Hub packets classify the hub as `HubIncoming`, `HubOutgoing`, or `HubBalanced`. They include fan counts, dominant direction, concentration measurements, representative caller or collaborator groups, and where useful, a compact local-graph summary.
- Layer-violation packets classify the violation as `LayerProjectReference` or `LayerNamespace`. They include the violated boundary and, for namespace-scope violations, the concrete crossing kinds, stored references, signature crossings, construction sites, and representative evidence.

Evidence sites are deliberately small. The packet builders select a few representative edges and turn them into `EvidenceSiteDto` records with an enclosing-member label, a line number, and a short snippet. When the source file can be read, the snippet is reconstructed as a short line window around the stored line number, giving the packet a small amount of surrounding source context. Otherwise, the stored snippet from the evidence entry is used. The packet therefore remains compact while still giving the consumer concrete code locations to cite.

7.4 Run Comparison and Architectural Drift

Run comparison is the reporting workflow for architectural change over time. It compares a reference run with a candidate run of the

same solution and produces a structured diff. Unlike inspection and recommendation, it works from `findings.summary.json` and `run-context.json`. It does not need the full report because it compares finding identities, severity values, metrics, baselines, and run scope.

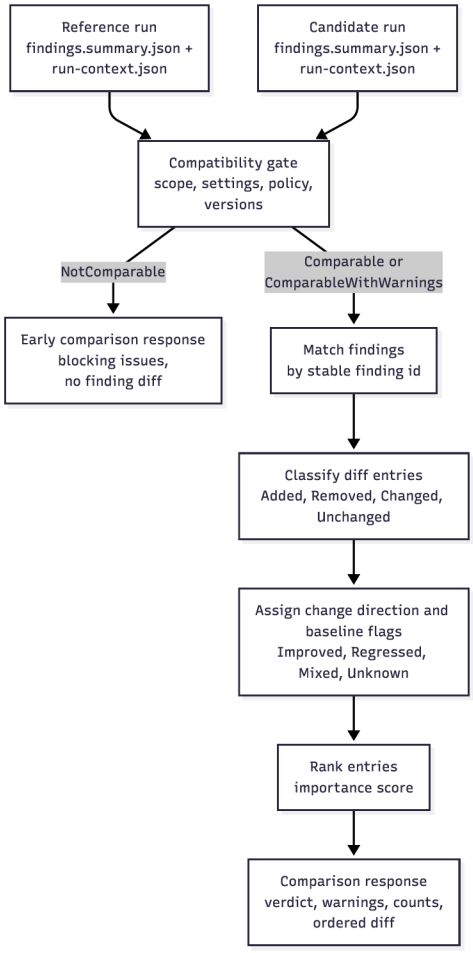

Figure 7.2: Run-comparison workflow from compatibility checking to ordered finding diff.

Compatibility gate. Before diffing findings, DotArchSentry checks whether the two runs were produced under comparable conditions. The gate returns `Comparable`, `ComparableWithWarnings`, or

`NotComparable`. A run is rejected as not comparable when changes in solution identity, extraction scope, analyzer configuration or layer policy would make the finding diff misleading. In that case, the response reports the blocking incompatibilities and does not produce a finding diff.

Warnings do not prevent comparison. They include changes in the included project or namespace scope, changes in graph-builder excluded project metadata, changes in blueprint drift, and changes in the baseline snapshot. In these cases the diff is still produced, but the response carries the warning list so the reader can interpret the result with the changed context in mind.

Finding diff. When the gate allows comparison, the diff service matches findings by stable finding identifier. A finding that appears only in the candidate run is `Added`. A finding that appears only in the reference run is `Removed`. A finding present in both runs is `Changed` when its severity or kind-specific metrics changed, and `Unchanged` otherwise.

Each diff entry also carries a trend field. Added findings are marked as `Regressed`, removed findings as `Improved`, and unchanged matched findings as `Unchanged`. For changed matched findings, the trend is derived from the severity delta and per-family metric directions. It can be `Improved`, `Regressed`, `Mixed`, or `Unknown`. For example, an increase in cycle member count, hub fan, or layer-violation crossing count is treated as a worsening signal.

Each diff entry receives an importance score used for ordering the comparison output:

$$\text{importance} = \text{base} + 0.5 \cdot \text{peakSeverity} + 1.0 \cdot |\text{severityDelta}|.$$

The base term is higher for added and removed findings than for changed findings, because the appearance or disappearance of a finding is usually more important to review than a small metric movement inside an existing finding. Importance is only a presentation ranking, it is not a second severity score.

Baseline and drift signals. Baseline membership is carried through comparison as two flags, `BaselinedInReference` and `BaselinedInCandidate`. The diff does not remove baselined findings, because a reviewer may still need to know that an accepted finding disappeared,

reappeared, or changed severity. Blueprint drift, introduced in Section 6.2, is surfaced through the compatibility gate as a warning when the drift items differ between the two runs. In that case, the saved policy may be stale, but that does not by itself prove that a dependency is invalid.

The comparison response therefore contains both the result and the conditions under which the result should be trusted: the compatibility verdict, blocking incompatibilities, warnings, scope changes, summary counts, and the ordered finding diff.

8 LLM Integration and Grounded Delivery

Chapters 5 to 7 describe how source code becomes a stream of findings and packets. This chapter describes the LLM layer that takes those packets and turns them into a conversation, on a local model, under scoping constraints that limit the model from authoring facts of its own.

8.1 Role and Boundaries of the LLM

By default, DotArchSentry runs the LLM on the developer's machine through a local Ollama instance. Two properties of the target setting motivate this. Source code in many deployments cannot be sent to external services because of non-disclosure agreements, regulation, or corporate policy. Cloud models also version silently, while a locally pinned tag carries a content-addressable digest that fixes the experimental setup in place. The hardware envelope follows directly: a 4 GB GPU caps the deployable model at four to seven billion parameters under 4-bit quantisation, with anything larger spilling to CPU at a few tokens per second.

Small local models hallucinate at higher rates than large cloud models, particularly when fed long structured inputs (§2.4). Such hallucinations appeared during the development of the bridge. On a packet describing a two-class cycle, a 4-billion-parameter model described the cycle as symmetric despite only one direction carrying construction, attributed a method to the wrong declaring class, and recommended a fix that left half of the cycle intact.

The conclusion is that architectural facts cannot be produced by the language model in this system. They must come from the analysis pipeline of Chapters 5 to 7; the model's job is only to phrase them. Structurally this is a retrieval-then-generation split, with the packet contract of Chapter 7 as the interface between the two halves. Anything inside a packet is the model's to phrase. Anything outside it (new strategy kinds, dependencies the extractor did not record, severity overrides, architectural claims beyond the current input) is not. The rest of the chapter develops the three mechanisms that enforce this boundary: the brief that shapes the input, the system prompt that

constrains the model's choices, and the retry gates that catch residual drift.

8.2 Grounded Briefs and Packet Presentation

A Grounded Brief is a markdown-style projection of a structured response returned by the API through an MCP tool. In the OllamaBridge, the presenter layer sits on the client side of the API boundary. The API returns JSON DTOs for programmatic consumers, while the bridge reshapes selected responses into prompt-oriented text for the local language model. The presenters do not change packet semantics and do not perform analysis. They are presentation adapters over API output.

Three bridge presenters cover the responses that need this treatment:

`FixOptionsBriefPresenter`	`get_finding_fix_options`
`FindingDetailBriefPresenter`	`inspect_finding_detail`
`RunComparisonBriefPresenter`	run-comparison responses

Each presenter renders the tool response as a short text brief. The brief begins with `[GROUNDED BRIEF]`, which lets the bridge recognize it later during context management. For recent inspection and fix-option calls, the original JSON is retained separately so that later checks can still read the full packet.

The packet DTOs described in Chapter 7 are shaped for API clients. Field names follow camelCase serialisation, structural relationships live in nested arrays, and important facts can sit below the top-level object. For example, the directionality of a cycle is stored in the `directions` array, and the interpretations addressed by a strategy are stored under each strategy's `addressesInterpretations` field. During bridge development, small local models were observed to overuse top-level prose fields and miss such nested structure. The brief is the bridge-side response to that failure mode: it surfaces the fields the model must preserve, especially dependency direction, finding shape, strategy scope, and representative evidence.

The contrast is visible on the `directions` array of `cycle:bdca25e3ee3c`. This example comes from a self-analysis run in which DotArch-

Sentry reported a cycle between `AgentTurn` and `BridgeShell` in its own OllamaBridge component. The relevant packet field is:

```
"directions": [
  {"fromMember": "OllamaBridge|type:AgentTurn",
   "toMember":   "OllamaBridge|type:BridgeShell",
   "dependencyKinds": ["Invocation"],
   "score": 24,
   "touchedMembers": ["RunWithStatusAsync","TrimForConsole"]},
  {"fromMember": "OllamaBridge|type:BridgeShell",
   "toMember":   "OllamaBridge|type:AgentTurn",
   "dependencyKinds": ["FieldType","Invocation","ObjectCreation"],
   "score": 21,
   "touchedMembers": ["RunAsync"]}
]
```

The fix-options presenter renders that direction data in a more direct form:

```
Direction summary:
- AgentTurn -> BridgeShell: invocation only
- BridgeShell -> AgentTurn: stores AgentTurn,
  constructs AgentTurn, and invokes AgentTurn

Directions:
- OllamaBridge|type:AgentTurn -> OllamaBridge|type:BridgeShell
  via Invocation (score 24); touches RunWithStatusAsync, TrimForConsole
- OllamaBridge|type:BridgeShell -> OllamaBridge|type:AgentTurn
  via FieldType, Invocation, ObjectCreation (score 21); touches RunAsync
```

The JSON representation requires the reader to assemble the asymmetry from two `dependencyKinds` arrays. The brief states the asymmetry first, then keeps the per-direction detail below it. In this example, the model sees that `AgentTurn` only invokes `BridgeShell`, while `BridgeShell` stores and constructs `AgentTurn`. That distinction matters because a fix recommendation should not describe construction or lifecycle ownership as mutual.

Rendering branches on the packet body kind, so cycle, hub, and layer-violation findings expose different fields to the model. Inspection presenters use the same body split but omit strategies, because inspection packets do not contain remediation options. Since the brief

is a lossy projection, the bridge also keeps the original packet JSON for recent inspection and fix-option calls; retry checks can then validate the model's answer against the full packet rather than the shortened prompt text.

8.3 System Prompt, Tool Routing, and Context Management

The OllamaBridge lets the model choose the next tool, but the bridge still controls how that tool call crosses the API boundary. This section describes the forward controls applied before and during a model turn: the bridge keeps the current solution state, completes common arguments, limits prompt history, and checks whether the model skipped a required tool call. Failures that slip past these forward controls are caught by the post-response checks of Section 8.4.

The first control point is the system prompt built by `ConversationContext.BuildInitialMessages`. Its routing part is a list of ordered workflow rules: analysis, solution-context setup, finding listing, finding inspection, fix-option requests, blueprint handling, run comparison, and ordinary no-tool answers. The prompt asks the model to choose the first matching rule. A fix request for a finding routes to `get_finding_fix_options`, an explanation request routes to `inspect_finding_detail`, and so on. These are prompt instructions rather than C# dispatch, so the bridge still needs runtime checks for cases where the model answers without using the required tool.

Beside the main system prompt, the bridge keeps a short known-state block that records the active solution path, the latest run identifier, and whether a previous tool result is waiting to be interpreted. This is what lets a user write `fix cycle:bdca25e3ee3c` without repeating the run identifier. Switching the active solution clears stale run and finding state so a finding from one solution is not reused for another.

Before a tool call reaches the MCP layer, `ToolArgumentNormalizer` rewrites the model's proposed arguments against a per-tool allowlist. Unsupported argument names are dropped, the active `solutionPath` and latest `runId` are filled in when the tool needs them, and placeholder-looking paths such as `path/to` or `your/solution` are rejected when a real active solution is known. The selected tool

still comes from the model; the bridge controls which arguments are allowed to reach the API.

Small local models do not always emit native tool calls in the expected shape. `TextualToolCallParser` handles this narrow failure mode by scanning the model's text for a compact JSON object with exactly `name` and `arguments` fields, including inside a fenced `json` block. Accepted objects are converted into the same internal tool-call shape as native calls. This fallback helps with formatting failures but does not parse arbitrary prose as a command.

After selected tools return, the bridge attaches a temporary post-tool instruction via `ToolInterpretationRules`. The instruction is specific to the tool that just ran and mirrors the constraints that Section 8.4 verifies. These messages steer the next response but are not stored as permanent conversation history. One workflow is handled more directly: after `run_solution_analysis` returns a run with findings, the bridge automatically calls `list_run_findings` without waiting for the model to ask. This fixed follow-up avoids a common small-model failure where the model reports that analysis has completed but forgets to retrieve the findings.

Prompt history is bounded. Before each model call the bridge prunes old conversation messages, cutting at user-message boundaries so the remaining prompt does not begin in the middle of a tool exchange. Stale Grounded Briefs are replaced by short breadcrumbs once a later user message exists, which keeps recent context dense without erasing the conversational thread. For retry prompts, the bridge can reduce the prompt to the system prompt, the known-state block, the latest brief, and the latest user message. Separately from this prompt history, the bridge maintains a small recent-packet cache holding the original JSON for inspection and fix-option packets, keyed by tool name and finding identifier. The cache is not part of what the model sees. It exists so retry checks can read the full packet after the visible prompt has been compacted away.

8.4 Grounding Controls and Retry Gates

Section 8.3 sets the model up to behave; this section catches it when it does not. The bridge runs checks over the model's response and

uses retry prompts for failures seen during development: answering without first retrieving a packet, narrating a tool call instead of executing it, ignoring a returned packet, inventing remediation strategies, or dropping cycle structure that was already present in the brief. These checks reduce common grounding failures but cannot prove the final answer is architecturally correct.

The checks are organized around three boundaries: tool use, packet interpretation, and recommendation scope. The first boundary is tool use. Some user requests require data from the API before an answer can be trusted. A request for finding details requires an inspection packet, and a request for fixes requires a recommendation packet. If the model answers such a request without calling the required tool, or merely narrates a call instead of executing it (saying it is running analysis, printing a JSON-looking call in prose, or promising to call later), the bridge treats the response as a tool-call formatting failure. It retries with a narrower instruction that includes the expected textual tool-call shape, which helps small models that fail to emit native tool calls but can still produce a compact JSON call. After two failed attempts the turn stops with an error rather than showing an ungrounded answer.

The second boundary is packet interpretation. After a tool has returned, the model is expected to explain the packet rather than ask the user what to do next. For example, after `get_finding_fix_options`, the model should explain the returned strategies instead of asking which option the user wants. The bridge retries this failure once with a direct instruction to interpret the packet. A related check catches long tutorial-like replies after a previous finding explanation or recommendation. These replies often contain commands, code blocks, or project-level refactoring advice that was not present in the packet. The retry asks for a shorter interpretive answer and rejects invented project or type names.

The third boundary is recommendation scope. Fix-option answers are checked more strictly because this is where the model is most likely to overreach. The answer must name a strategy kind or title that appears in the returned packet. If the packet contains method slices or representative evidence sites, the answer must cite at least one of them. The answer must also respect each strategy's own `addressesInterpretations` field. A strategy that addresses only

`StateCoupled`, for example, must not be described as also addressing `ConstructionDriven`. When this check fails, the bridge retries with a reduced prompt containing only the latest brief, the latest user message, and the required instruction. If the repeated answer still fails the same packet-consistency check, the answer is printed with a visible `[GROUNDING WARNING]` block. The warning tells the user that the packet fields remain authoritative.

Strategy names have a separate guard. Each recommendation packet contains `allowedStrategyKinds`, the exact set of remediation kinds that may be named for that finding. If the model names a strategy outside that set, the bridge retries once with the allowed set repeated. If the second answer still contains an unauthorized strategy name, that name is replaced with `[unauthorized strategy redacted]` before the answer is printed. The bridge does not rewrite the recommendation. It only removes a strategy kind that crossed the packet boundary.

One final repair protects cycle shape. Cycle packets can contain a representative closed path, a mesh structure, or a fallback member block. If a fix-option answer omits that structure, the bridge prepends the corresponding block from the latest Grounded Brief. This does not add new analysis. The copied structure still comes from the packet-derived brief, and no new edge is inferred by the bridge. The repair exists because small models can summarize a cycle in vague prose even when the packet already contains the concrete shape. The bridge also marks answers that are no longer backed by a fresh brief in the current prompt. If the user continues a discussion after the original brief has been compacted away, the bridge may prepend an interpretive note. The answer is still allowed, but the note tells the user that the guidance is less directly tied to current tool output.

These controls are intentionally limited. They rely on packet fields, brief markers, and string checks over the model's answer. They cannot prove that a proposed refactoring is the best engineering choice, and they cannot compensate for missing or incorrect data in the packet. Their role is narrower: require packet retrieval when a packet is needed, retry common interpretation failures, make unresolved consistency failures visible, and keep the final answer inside the strategy and evidence boundary supplied by the reporting pipeline. Chapter 9 evaluates how much these controls improve local-model answers in practice.

9 Evaluation

This chapter checks two parts of DotArchSentry. The first is the static-analysis pipeline: whether the extractor and analyzers produce architectural findings that can be defended against source code and an external tool. The second is the local-LLM delivery path: whether raw packets, rendered briefs, and retry guards change how faithfully a model explains the same fix-options packet.

The detailed run artifacts are submitted with the source attachment under `evaluation/`. Appendix A lists the files and their role.

9.1 Evaluation Design

9.1.1 Corpus

The evaluation uses four C# solutions. `DotArchSentryStressLab` is a synthetic solution written for this thesis. It contains known cycle, hub, god-class, interface-centrality, and project-layer-violation shapes, so it acts as a positive control. DotArchSentry self-analysis runs the tool against its own solution and is also the target of the NDepend comparison. StockHustlerBE and MusicoStore are external course projects without labelled architectural ground truth; they are used as plausibility checks, where a finding counts only if its evidence can be traced to source and its interpretation makes sense.

9.1.2 Run Conditions

All DotArchSentry runs used the default sensitivity profile and `internal` extraction depth. Layer-violation analysis was exercised only on StressLab, whose saved project-scope blueprint forbids `Stress.Application` → `Stress.Infrastructure`. The other three solutions were run without saved blueprints, so their layer analyzers were not instantiated.

The run-comparison workflow was tested on two StressLab runs: `run-20260515185227-2cf9af50` as the reference and `run-20260515225616-23bd9422` as the controlled candidate. The candidate added one collaborator to a god-class fixture, one caller of an existing hub,

one dependency inside a mutual-pair cycle, and a new project not covered by the saved blueprint. NDepend v2026.1.5 was used for the self-analysis cross-check. The comparison enabled rule ND1409 for mutually dependent types and used a CQLinq query to retrieve size, coupling, and cohesion metrics. The query and exports are stored under `evaluation/NDepend`.

The LLM harness measured only the fix-options workflow. It read ten persisted packets from disk, eight from StressLab and two from self-analysis, and invoked locally hosted Ollama models with `temperature=0`, `num_predict=1024`, and `think=false`. Each row built a fresh four-message request, so no conversation state carried between rows. The sampled models were `qwen3.5:0.8b`, `qwen3.5:2b`, and `qwen3.5:4b`. The prompt configurations were `raw-only`, `brief-only`, and `brief-plus-guards`; `num_ctx=4096` was the production setting and `num_ctx=2048` was the stress point. The two context settings produce 180 rows in total. Section 9.3.1 describes the rubric and manual audit applied to those rows.

9.2 Static Architecture Analysis Results

9.2.1 StressLab: Designed Findings and Run Comparison

The StressLab run at `internal` depth produced the expected eight findings: three cycle shapes, two implementation hotspots, one interface-centrality hub without fix strategies, one god-class candidate, and one project-scope layer violation. This is a positive-control result, not a general accuracy claim.

The controlled comparison between the two StressLab runs was classified as comparable with warnings. No findings were added or removed; four findings changed and four remained unchanged. The changed findings were the expected ones: `SolutionAnalysisCoordinator`, `NotificationRouter`, `ICoreFormatter`, and the mutual-pair cycle `cycle:9dc6d3eda5a1`. The candidate run also produced blueprint drift on `Stress.AddedProject`. The full diff is preserved in the submitted run artifacts.

9.2.2 Self-Analysis and NDepend Cross-Check

The self-analysis run (`run-20260515185155-e4a8ca59`) produced two findings: a god-class candidate on `AnalysisApplicationService` and a mutual-pair cycle between `AgentTurn` and `BridgeShell`. Passive data-carrier types such as `Node`, `Edge`, `Finding`, and `DependencyGraph` did not appear as findings.

Table 9.1: Cross-tool comparison on the DotArchSentry self-analysis codebase.

Comparison point	NDepend result	DotArchSentry result
Type-level cycle	ND1409 reported `AgentTurn` ↔ `BridgeShell` and two compiler-emitted top-level-statement helper pairs.	Reported the developer-visible `AgentTurn` ↔ `BridgeShell` cycle only.
God-class candidate	The CQLinq query found `DotArchSentry.Api.Services.AnalysisApplicationService` with `TypesUsed`=123, LCOMHS = 0.89, LCOM = 0.85, 20 methods, 9 fields, and 160 LoC.	Reported `AnalysisApplicationService` as the sole self-analysis god-class candidate, with severity 0.9750.

The reported LCOMHS value corresponds to NDepend's Henderson-Sellers lack-of-cohesion metric [29]. The comparison is intentionally narrow and should not be read as treating NDepend as a precision-and-recall oracle, since the tools use different rule catalogues and reporting models. Nevertheless, it provides a useful external check: an independent .NET analysis tool identifies the same developer-visible dependency cycle and the same oversized orchestration class.

9.2.3 External Course Projects

StockHustlerBE (`run-20260518171021-c4adfa39`) produced three findings: outgoing hub candidates on `DatasetService` and `DatasetReportHandler`, plus a localized `EmailSettings` ↔ `EmailSettingsV`

alidator cycle. Manual inspection supports the two hubs as plausible service-level hotspots. DatasetService stores IUnitOfWork, IMapper, and IDatasetParser; its insert workflow parses an uploaded dataset, resolves or creates companies, and writes dataset and holding entities through repositories. DatasetReportHandler coordinates dataset retrieval, difference construction, notification creation, report formatting, and batch email sending through five collaborators.

The settings-validator cycle is also real but local. EmailSettings.Validate constructs EmailSettingsValidator, while EmailSettingsValidator is typed over EmailSettings and reads its properties through validation rules.

MusicoStore (run-20260516183916-34167b80) produced four findings: an AdminController god-class, two interface-centrality findings on generic service/repository abstractions, and a localized DataInitializer ↔ StoreDbContext cycle. The AdminController finding is supported by 52 distinct outgoing evidence targets and 41 methods across workflows such as orders, gift cards, product audits, and category administration. The two interface-centrality findings were treated as neutral structural signals rather than defects, because generic repository and service interfaces are expected coordination points in this application.

9.3 LLM-Assisted Interpretation Results

9.3.1 Harness, Rubric, and Manual Semantic Audit

As part of the evaluation tooling, an ablation harness was implemented to replay persisted fix-options packets against local Ollama models under controlled prompt configurations. The harness produced a packet-response matrix and assigned each response a five-item structural rubric score. This rubric captured surface-level properties, including the use of sanctioned strategy names, whether the production guard passed or failed, the presence of cited evidence, the absence of unsanctioned strategy names, and agreement with the expected finding shape. It was useful for initial triage, but it could not fully assess semantic correctness. A response may cite the expected evidence while still reversing the direction of a dependency cycle or overstating architectural dominance.

For that reason, all 180 rows were manually audited against the corresponding source packet. The manual audit uses a 0–5 scale: 0 denotes context collapse, 1 misleading or hallucinated output, 2 major omissions or errors, 3 partially correct output, 4 usable output with a concrete issue, and 5 a response that is faithful to the packet in strategy, interpretation, and evidence.

9.3.2 Results at the 4096-Token Default

At `num_ctx=4096`, the main pattern is model size. The 0.8b and 2b models stayed between 2.50 and 3.20 mean audit score across all prompt configurations and produced no fully faithful 5/5 rows. The 4b model was usable across all three prompt forms: raw-only reached 4.20 (3/10 fully faithful), brief-only 4.30 (4/10), and brief-plus-guards 4.10 (3/10). At this context size, the rendered brief and guards did not clearly outperform raw JSON. The larger model could read the packet directly well enough that the presentation layer no longer gave a consistent advantage.

9.3.3 The 2048-Token Stress Point

The 2048-token setting is where the configurations separate.

Table 9.2: Manual audit results at the 2048-token stress point, aggregated across the three model sizes.

	Context collapses	Mean audit score	Fully faithful
`raw-only`	27/30	0.30	0/30
`brief-only`	15/30	1.63	3/30
`brief-plus-guards`	0/30	3.23	5/30

All 42 collapsed rows ask for a `.sln` path instead of engaging with the packet. The pattern is consistent with input truncation: the routing prompt survives, while the packet and user request are no longer available to the model. The guarded configuration avoids this failure because the retry path rebuilds a shorter prompt from the latest brief and known state. At `num_ctx=2048`, `brief-plus-guards` retried 18 of 30 rows. All 18 retries were grounding retries; in two of those

rows, the strategy-scope retry flag was also set. The supported claim is specific: briefs and guarded retries rescue the fix-options workflow in the fragile setting of small context windows and nested structured packets.

9.3.4 Observed Failure Modes

The manual audit surfaced five recurring failures: context collapse, invented dominance, direction inversion on cycle packets, strategy-name slips, and leaked cycle-section labels on hub or layer-violation packets. One representative case is `ctx4096|qwen3.5:2b|raw-only|cycle:bdca25e3ee3c`, where the response reverses which type stores the peer in a mutual-pair cycle. For row-level detail, the submitted audit viewer listed in Appendix A provides the packet, model response, and audit note for each row. These failures support the reporting boundary from Chapter 8: the model can explain stored findings, but the architectural facts must remain in analyzer output and packet contracts.

9.4 Threats to Validity

Corpus. StressLab checks designed shapes, not unexpected architecture. For StockHustlerBE and MusicoStore, there was no pre-existing list of correct architectural findings to compare against. The reported findings were therefore checked against source evidence, which is weaker than evaluation on a benchmark with expected results.

Cross-tool comparison. NDepend was used as a second opinion for the self-analysis cycle and god-class findings. The layer-violation analyzer was exercised on StressLab, but its results were not compared against architecture-testing frameworks such as ArchUnitNET.

Manual audit. The LLM responses were scored by one evaluator, so the audit reflects one person's judgement of whether a response is faithful to the packet. The submitted CSV and viewer make the decisions inspectable, but they do not make the scores definitive. A stronger evaluation would use multiple independent evaluators and compare their agreement.

LLM coverage. The harness evaluates only fix-options responses, ten packets, three local model tags, two context windows, and one

deterministic run per cell. Inspection and run-comparison briefs are implemented but not measured by this harness.

Heuristic calibration. The severity weights from Section 6.4 are fixed heuristics. They were not calibrated against labelled architectural severity judgments.

10 Conclusion

10.1 Summary of Contributions

This thesis presented DotArchSentry, a source-derived architecture analysis tool for C# solutions. The system loads a solution through Roslyn, extracts a type-level dependency graph, detects selected architectural findings, stores the resulting artifacts, and lets a local LLM explain those artifacts without deciding which facts exist. The work makes four contributions.

The first contribution is the extractor and graph model. DotArchSentry records declaration-level dependencies and selected body-level dependencies, including source file, line number, snippet, enclosing member, and edge metadata where later reporting needs it.

The second contribution is the analyzer family. Cycle, hub, namespace-scope layer-violation, and project-scope layer-violation analyzers emit findings under one artifact contract. Findings also carry kind-specific structure: hubs distinguish interface centrality, implementation hotspots, and god-class candidates, while cycles carry shape and interpretation data.

The third contribution is the reporting pipeline. Findings are transformed into inspection and fix-options packets through fact extraction, interpretation, and strategy selection. Run comparison reads persisted summaries and run context, then reports compatibility, changed findings, trends, baseline status, and blueprint drift.

The fourth contribution is the delivery layer. The HTTP API exposes the workflows directly, the MCP server forwards selected workflows as tools, and the OllamaBridge presents results through a local Ollama model. The bridge uses briefs and retry gates as best-effort controls to keep responses tied to packet contents and to reduce unsupported strategy names in recommendations.

The evaluation supports these contributions. StressLab recovered the designed finding shapes and exercised run comparison. Self-analysis found the `AnalysisApplicationService` god-class candidate and the `AgentTurn` $\leftrightarrow$ `BridgeShell` cycle, both also identified by the NDepend cross-check. StockHustlerBE and MusicoStore served as external plausibility checks rather than benchmark evidence: their

findings could be traced to source, but some were neutral structural signals rather than defects, such as MusicoStore's generic interface-centrality findings. The LLM harness showed the boundary of the local delivery design: at `num_ctx=2048`, raw API JSON packets collapsed on 27 of 30 rows under context pressure, while `brief-plus-guards` avoided this collapse across all 30 rows and produced usable responses more consistently.

10.2 Limitations and Future Work

DotArchSentry remains a static-analysis tool. Mechanisms such as reflection, runtime dependency injection, and dynamic loading can create relationships that are not fully visible in Roslyn's source-level model. Future work could combine the graph with build logs or runtime traces, adding a dynamic-analysis component for dependencies that are not visible from source alone.

The evaluation should also be expanded. Future work should use more external C# systems, include expected findings prepared independently from DotArchSentry, and compare more analyzer families against existing tools, including layer-rule checks against frameworks such as ArchUnitNET. The LLM evaluation should also cover inspection and run-comparison briefs, more model families, and multiple independent evaluators for the semantic audit.

The analyzer heuristics also need calibration. The current thresholds and severity weights are fixed rules chosen for this prototype. Future work should tune them against a larger set of reviewed findings, distinguish composition roots more explicitly from ordinary hubs, and refine the ranking of findings by architectural importance.

Finally, the file-based persistence model and local deployment need maintenance support. During evaluation, manual deletion of run directories left stale entries in the global run index. A production version should add index repair or garbage collection that reconciles the index with the run directories present on disk. It should also simplify startup, for example through a packaged executable, Docker-based setup, or another distribution form that hides the current multi-process launch sequence.

Bibliography

1. SILVA, Lakruwan de; BALASUBRAMANIAM, Dharini. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*. 2012, vol. 85, no. 1, pp. 132–151. Available from DOI: 10.1016/j.jss.2011.07.036.
2. SHARIR, M. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*. 1981, vol. 7, no. 1, pp. 67–72. ISSN 0898-1221. Available from DOI: 10.1016/0898-1221(81)90008-0.
3. MARINESCU, Radu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2004, pp. 350–359. Available from DOI: 10.1109/ICSM.2004.1357820.
4. LANZA, Michele; MARINESCU, Radu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin, Heidelberg: Springer, 2006. ISBN 9783540244295. Available from DOI: 10.1007/3-540-39538-5.
5. BRIAND, Lionel C.; DALY, John W.; WÜST, Jürgen. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*. 1999, vol. 25, no. 1, pp. 91–121. Available from DOI: 10.1109/32.748920.
6. MICROSOFT. *The .NET Compiler Platform SDK* [online]. Microsoft Learn. [visited on 2026-05-12]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.
7. MICROSOFT. *Work with syntax* [online]. Microsoft Learn. [visited on 2026-05-12]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-syntax.
8. MICROSOFT. *Get started with semantic analysis* [online]. Microsoft Learn. [visited on 2026-05-12]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/semantic-analysis.

9. MICROSOFT. *Work with a workspace* [online]. Microsoft Learn. [visited on 2026-05-12]. Available from: `https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-workspace`.

10. CHUNG, Hyung Won; HOU, Le; LONGPRE, Shayne; ZOPH, Barret; TAY, Yi; FEDUS, William; LI, Yunxuan; WANG, Xuezhi; DEHGHANI, Mostafa; BRAHMA, Siddhartha; WEBSON, Albert; GU, Shixiang Shane; DAI, Zhuyun; SUZGUN, Mirac; CHEN, Xinyun; CHOWDHERY, Aakanksha; CASTRO-ROS, Alex; PELLAT, Marie; ROBINSON, Kevin; VALTER, Dasha; NARANG, Sharan; MISHRA, Gaurav; YU, Adams; ZHAO, Vincent; HUANG, Yanping; DAI, Andrew; YU, Hongkun; PETROV, Slav; CHI, Ed H.; DEAN, Jeff; DEVLIN, Jacob; ROBERTS, Adam; ZHOU, Denny; LE, Quoc V.; WEI, Jason. *Scaling Instruction-Finetuned Language Models*. 2022. Available from arXiv: `2210.11416 [cs.LG]`.

11. OLLAMA. *Ollama* [online]. Ollama. [visited on 2026-05-12]. Available from: `https://github.com/ollama/ollama`.

12. JI, Ziwei; LEE, Nayeon; FRIESKE, Rita; YU, Tiezheng; SU, Dan; XU, Yan; ISHII, Etsuko; BANG, Ye Jin; MADOTTO, Andrea; FUNG, Pascale. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 2023, vol. 55, no. 12. ISSN 0360-0300. Available from DOI: `10.1145/3571730`.

13. MAYNEZ, Joshua; NARAYAN, Shashi; BOHNET, Bernd; MCDONALD, Ryan. On Faithfulness and Factuality in Abstractive Summarization. In: JURAFSKY, Dan; CHAI, Joyce; SCHLUTER, Natalie; TETREAULT, Joel (eds.). *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, 2020, pp. 1906–1919. Available from DOI: `10.18653/v1/2020.acl-main.173`.

14. LEWIS, Patrick; PEREZ, Ethan; PIKTUS, Aleksandra; PETRONI, Fabio; KARPUKHIN, Vladimir; GOYAL, Naman; KÜTTLER, Heinrich; LEWIS, Mike; YIH, Wen-tau; ROCKTÄSCHEL, Tim; RIEDEL, Sebastian; KIELA, Douwe. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. Vancouver, BC, Canada: Curran Associates Inc., 2020. NIPS '20.

ISBN 9781713829546. Available also from: https://dl.acm.org/doi/abs/10.5555/3495724.3496517.

15. LIU, Nelson F.; LIN, Kevin; HEWITT, John; PARANJAPE, Ashwin; BEVILACQUA, Michele; PETRONI, Fabio; LIANG, Percy. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*. 2024, vol. 12, pp. 157–173. Available from DOI: 10.1162/tacl_a_00638.

16. SONARSOURCE. *SonarQube* [online]. SonarSource. [visited on 2026-05-14]. Available from: https://www.sonarsource.com/products/sonarqube/.

17. NDEPEND. *NDepend Features* [online]. NDepend. [visited on 2026-05-14]. Available from: https://www.ndepend.com/features.

18. TNG TECHNOLOGY CONSULTING. *ArchUnitNET* [online]. GitHub. [visited on 2026-05-14]. Available from: https://github.com/TNG/ArchUnitNET.

19. MORRIS, Ben. *NetArchTest.Rules* [online]. NuGet Gallery. [visited on 2026-05-14]. Available from: https://www.nuget.org/packages/NetArchTest.Rules.

20. FONTANA, Francesca Arcelli; PIGAZZINI, Ilaria; ROVEDA, Riccardo; TAMBURRI, Damian; ZANONI, Marco; DI NITTO, Elisabetta. Arcan: A Tool for Architectural Smells Detection. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 282–285. Available from DOI: 10.1109/ICSAW.2017.16.

21. ARCAN. *Architectural Smells* [online]. Arcan. [visited on 2026-05-14]. Available from: https://docs.arcan.tech/latest/architectural_smells/.

22. YAO, Shunyu; ZHAO, Jeffrey; YU, Dian; DU, Nan; SHAFRAN, Izhak; NARASIMHAN, Karthik; CAO, Yuan. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. Available from arXiv: 2210.03629 [cs.CL].

23. SCHICK, Timo; DWIVEDI-YU, Jane; DESSÌ, Roberto; RAILEANU, Roberta; LOMELI, Maria; ZETTLEMOYER, Luke; CANCEDDA, Nicola; SCIALOM, Thomas. *Toolformer: Language Models Can Teach Themselves to Use Tools*. 2023. Available from arXiv: 2302.04761 [cs.CL].

24. MODEL CONTEXT PROTOCOL. *What is the Model Context Protocol (MCP)?* [online]. LF Projects. [visited on 2026-05-13]. Available from: https://modelcontextprotocol.io/docs/getting-started/intro.

25. MICROSOFT. *Entity Types* [online]. Microsoft Learn. [visited on 2026-05-14]. Available from: https://learn.microsoft.com/en-us/ef/core/modeling/entity-types.

26. MICROSOFT. *Relationship navigations* [online]. Microsoft Learn. [visited on 2026-05-14]. Available from: https://learn.microsoft.com/en-us/ef/core/modeling/relationships/navigations.

27. MARTIN, Robert C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall, 2002. ISBN 9780135974445.

28. FOWLER, Martin; BECK, Kent; BRANT, John; OPDYKE, William; ROBERTS, Don. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN 9780201485677.

29. HENDERSON-SELLERS, Brian. *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ: Prentice Hall, 1996. ISBN 9780132398725.

A Evaluation Artifacts

The full evaluation material referenced in Chapter 9 is submitted with the source attachment under `evaluation/`. This appendix lists the files and the role each one plays in supporting the chapter's claims. The directory also contains a `README.md` that documents the harness commands needed to reproduce the raw rows.

Static-Analysis Evidence

`static-runs/` holds persisted DotArchSentry run artifacts grouped by codebase. Each leaf directory contains `metadata.json`, `report.full.json`, `findings.summary.json`, and `run-context.json`. The subdirectories are:

- `stresslab/` – the two StressLab runs used for the run-comparison example in Section 9.2.1, plus a third run kept as a comparison-incompatible example.
- `dotarchsentry-self/` – the self-analysis run cited in Section 9.2.2.
- `stockhustlerbe/` – the StockHustlerBE run from Section 9.2.3.
- `musicostore/` – the MusicoStore run from Section 9.2.3.

`NDepend/` holds the cross-tool comparison exports cited in Section 9.2.2:

- `AgentTurn-Bridge-cycle-ndepend-rule-ND1409.csv` – the ND1409 cycle report on the self-analysis solution.
- `DotArchSentry-analysis-godclass.csv` – the CQLinq query result for the god-class cross-check.
- `query-for-godclass.txt` – the CQLinq query text itself.

LLM Evaluation Evidence

The four CSV files plus the read-only HTML viewer carry the data that supports Section 9.3.

- `stresslab-plus-real-thesis-configs-ctx2048.csv` – raw harness rows at `num_ctx`=2048, one row per $(finding \times config \times model)$ cell, with the harness's structural rubric score, retry flags, response length, elapsed wall time, and the full response text.
- `stresslab-plus-real-thesis-configs-ctx4096.csv` – the same shape at `num_ctx`=4096.
- `stresslab-plus-real-thesis-configs_manual-audit.csv` – the manual semantic audit described in Section 9.3.1. One row per `row_id` with columns `audit_score` (0–5), `audit_verdict` (label), and `audit_note` (justification). This file is the source for the cell values in Section 9.3.2 and Table 9.2.
- `stresslab-plus-real-thesis-configs_audit-payloads.csv` – the packet JSON sent to the model and the model's response, joined by `row_id`. The source of truth that lets a reader open any audited row and see both inputs and output.
- `stresslab-plus-real-thesis-configs_audit-viewer.html` – a read-only browser viewer that embeds the payload CSV and the manual-audit CSV inline. Opening it in any modern browser shows every row's packet, model response, and audit verdict side by side. It is a static snapshot at submission time, not a dynamic source.

B Running the Submitted Application

The submitted attachment contains the DotArchSentry source repository under `source/`. That repository contains the solution file and a repository-level `README.md` with detailed build and run instructions. The submitted source repository includes the MIT License file. The application targets `net8.0` and was developed and evaluated on Windows with the .NET 8 SDK and a local Ollama runtime.

To run the interactive local-LLM workflow, the following components are required:

- .NET 8 SDK, available from `https://dotnet.microsoft.com/download/dotnet/8.0`,
- Ollama, available from `https://ollama.com/download`, running on `http://localhost:11434` (by default),
- the `qwen3.5:4b` model for the default OllamaBridge setup.

The repository includes a PowerShell startup script:

```
powershell -ExecutionPolicy Bypass -File .\scripts\Start-OllamaBridge.ps1
```

The script checks the required tools, starts the DotArchSentry API and OllamaBridge, and prints status information. The README also describes manual startup, MCP configuration through `.mcp.json`, and the commands used to reproduce the LLM harness rows.