Automated Implementation of Architectural Tactics for Software Quality Improvement

A Comprehensive Study Guide

Innopolis University — MS Study Notes, 2026

February 2026

Contents

Preface

This study guide accompanies the thesis *Automated Implementation of Architectural Tactics for Software Quality Improvement* (Innopolis University, 2026). It is designed as a self-contained educational resource that walks the reader through the foundational concepts, research landscape, and open problems at the intersection of software architecture, maintainability, and large language models.

Who this guide is for. Graduate students (MS level) studying software architecture and software quality engineering. It assumes you have completed an undergraduate course in software engineering and are comfortable reading code in at least one mainstream programming language (Python, Java, or similar). No prior expertise in architectural tactics, static analysis, or LLMs is required — the guide builds these concepts from the ground up.

Prerequisites. A working understanding of basic software engineering concepts: modules, interfaces, coupling, cohesion, version control, and unit testing. Familiarity with at least one object-oriented or multi-paradigm programming language. Exposure to design patterns (e.g., Observer, Facade, Strategy) is helpful but not mandatory.

How to use this guide. The chapters are ordered so that each one builds on the preceding material. If you are new to the topic, read sequentially from Chapter 1 (Motivation) through to the end. If you already have a background in software architecture, you may jump directly to Chapter 4 (Architectural Tactics) or Chapter 7 (LLMs for Code Refactoring). Each chapter opens with learning objectives and closes with review questions to test your understanding.

A note on citations. All references in this guide use the format `[@key]` and resolve to the bibliography at the end of the document. For example, `[@bass2021software]` refers to Bass, Clements, and Kazman's *Software Architecture in Practice* (4th ed., 2021). When multiple sources support a single claim, they appear together: `[@perry1992foundations; @garlan1993introduction]`. The full bibliography is provided in `references.bib` and can be processed with pandoc-citeproc.

Chapter 1

Motivation and Context

Learning objectives. After reading this chapter you should be able to (1) explain why software maintenance dominates lifecycle costs, (2) identify the knowledge barriers that prevent developers from applying architectural tactics, (3) articulate the promise — and the current limits — of using LLMs for architecture-level code improvement, and (4) describe the scope and structure of this study guide.

1.1 The Software Maintenance Crisis

Software is not built once and forgotten. The overwhelming majority of a system's total cost of ownership is spent not on initial development but on understanding, modifying, extending, and fixing the code that already exists. Industry data consistently places maintenance at **60–80% of total lifecycle costs** [1]. Martin Fowler goes further, estimating that up to **75% of development effort** is consumed by refactoring — the restructuring of existing code to improve its internal quality without changing its external behavior [2].

Why is maintenance so expensive? The root cause is seldom the individual line of code. Instead, it is the *decisions made at the architectural level* — the choice of decomposition strategy, the way modules communicate, the degree to which responsibilities are encapsulated — that determine how easily (or painfully) a system can absorb future changes. Garlan and Perry captured this insight in a memorable metaphor:

> "Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the **load-bearing walls** of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications." [3]

Just as removing a load-bearing wall in a building risks structural collapse, modifying a core architectural decision in software can trigger cascading changes across dozens of files, modules, or services. Conversely, a well-chosen architecture confines the blast radius of change. The maintenance crisis, then, is fundamentally an *architecture* crisis.

Cost factor	Typical range	Source
Maintenance share of total lifecycle cost	60–80%	[1]
Development effort spent on refactoring	up to 75%	[2]
Practitioners reporting quality degradation from architecture erosion	83.8%	[4]

1.2 The Knowledge Barrier

If good architectural decisions are the key to maintainability, why do so many systems end up poorly structured? The answer lies in a persistent knowledge barrier that separates *knowing what to do* from *knowing how to do it in code.*

1.2.1 Cross-cutting tactics require deep framework knowledge

Architectural tactics — the fine-grained design decisions that target a single quality attribute [1] — are conceptually straightforward. "Use an intermediary to decouple producers from consumers" is easy to state. But *implementing* that tactic in a real codebase often requires touching multiple files, understanding framework-specific APIs, and coordinating changes across method boundaries. Shokri et al. demonstrated this concretely: implementing a JAAS authentication tactic (a security tactic) requires inter-procedural changes spanning multiple classes and packages, with API calls that must be placed in precisely the right locations [5]. When they tested ChatGPT on this task, the LLM produced syntactically correct code 95% of the time — but only **5% of implementations were semantically correct**, because the model failed to coordinate changes across methods and classes.

1.2.2 Novice developers produce measurably worse code

The knowledge barrier is especially acute for less experienced developers. Haindl and Weinberger conducted a controlled experiment comparing code written by novice programmers with and without ChatGPT assistance. The ChatGPT-assisted group produced code with significantly lower cyclomatic complexity and fewer coding convention violations ($p < 0.005$) [6]. This is encouraging at the code level, but it also reveals a gap: current LLM tools help with *local* code quality (naming, complexity, style) but do not address *architectural* quality (module boundaries, coupling patterns, tactic implementation).

1.2.3 Architecture erosion: the silent degradation

Even when systems start with a well-designed architecture, implementations gradually diverge from the intended design — a phenomenon formally described by Perry and Wolf as **architectural erosion** (violations of architecture leading to brittleness) and **architectural drift** (insensitivity to architecture leading to loss of coherence) [7].

Li et al. conducted a systematic mapping study of 73 studies on architecture erosion and found that **83.8% of practitioners report quality degradation** as a direct consequence [4]. The top technical causes include architecture violations (24.7%), evolution issues (23.3%), and technical debt (17.8%). Critically, technical debt forms a *vicious cycle* with erosion: it is both a cause and a consequence.

Rosik et al. provided striking empirical evidence in a 2-year longitudinal case study at IBM: architectural drift occurred even during initial de novo implementation, even when the architect was also the sole developer [8]. More troublingly, **identifying drift did not lead to its removal** — developers tolerated violations because fixing them risked ripple effects, consumed time, and entangled legacy code. Detection alone is not enough; developers need low-cost, low-risk *remediation* paths.

Barrier	Evidence	Source
Cross-cutting tactic implementation requires inter-procedural coordination	ChatGPT: 95% syntax-correct but only 5% semantically correct on tactic synthesis	[5]
Novice developers produce higher-complexity code	Controlled study: ChatGPT reduces cyclomatic and cognitive complexity ($p < 0.005$)	[6]
Architecture erosion degrades quality	83.8% of practitioners report quality degradation	[4]
Drift occurs even with a single architect-developer	9 divergent edges found in 2-year IBM case study	[8]
Detection does not equal remediation	Zero identified inconsistencies were removed by developers	[8]

1.3 The Promise of LLMs

Large language models have demonstrated remarkable ability to understand and transform source code. In the domain of refactoring — the restructuring of code to improve quality without changing behavior — recent results are striking:

- **MANTRA** [9], a multi-agent LLM framework, achieved an **82.8% success rate** (582 out of 703 cases) in producing compilable, test-passing refactored Java code across six refactoring types. A baseline single-prompt LLM achieved only 8.7%.
- A user study with 37 developers found MANTRA-generated code *comparable to human-written code* in readability (4.15 vs. 4.02) and reusability (4.13 vs. 3.97), with no statistically significant difference.
- The ablation study revealed that the **Reviewer Agent** — which uses traditional SE tools (RefactoringMiner, CheckStyle) to provide structured feedback — was the most critical component, contributing a 61.9% improvement. This suggests that combining LLMs with static analysis verification is more effective than using LLMs alone.

These results establish a crucial precedent: LLMs can perform code-level transformations reliably, *provided* they operate within a structured pipeline that includes external verification. But current tools operate almost exclusively at the **code level** — extracting methods, inlining variables, moving classes. They do not reason about **architecture-level** concerns: module boundaries, coupling patterns, quality attribute trade-offs, or the systematic application of architectural tactics.

This is the gap that motivates the present work. Can LLMs bridge the distance between *design intent* (an architectural tactic specification) and *code implementation* (the actual changes across files, classes, and methods)? Can we build a pipeline that takes a tactic like "Use an Intermediary" and automatically produces the correct multi-file transformation in a real codebase — while preserving behavior, respecting existing architecture, and improving measurable maintainability?

1.4 A Motivating Example

Consider a Python web service for an e-commerce platform. The service handles product catalog queries, and every endpoint directly queries the database:

```python
# routes/products.py
from db import get_connection

def get_product(product_id):
    conn = get_connection()
    result = conn.execute("SELECT * FROM products WHERE id = ?", (product_id,))
    return result.fetchone()

def get_products_by_category(category_id):
    conn = get_connection()
    result = conn.execute(
        "SELECT * FROM products WHERE category_id = ?", (category_id,)
    )
    return result.fetchall()
```

```python
# routes/orders.py
from db import get_connection

def get_order_products(order_id):
    conn = get_connection()
    result = conn.execute(
        "SELECT p.* FROM products p JOIN order_items oi ON p.id = oi.product_id "
        "WHERE oi.order_id = ?", (order_id,)
    )
    return result.fetchall()
```

```python
# routes/recommendations.py
from db import get_connection

def get_similar_products(product_id):
    conn = get_connection()
    # Complex query: get products in same category, ordered by popularity
    result = conn.execute(
        "SELECT p.* FROM products p WHERE p.category_id = "
        "(SELECT category_id FROM products WHERE id = ?) "
        "ORDER BY p.view_count DESC LIMIT 10", (product_id,)
    )
    return result.fetchall()
```

Now a change request arrives: **"Add caching to reduce database load for product queries."**

In the current architecture, every module that touches product data has its own direct database call. Implementing caching requires modifying *every call site* — potentially **15 or more files** across routes, background workers, admin endpoints, and test fixtures. Each modification risks inconsistency: one file might cache with a 5-minute TTL, another with 10 minutes, and a third might forget caching entirely.

Applying the "Use an Intermediary" tactic transforms this situation. We introduce a single ProductRepository that encapsulates all product data access and add caching at that layer:

```
# repositories/product_repository.py
from db import get_connection
from cache import cache

class ProductRepository:
    @cache(ttl=300)
    def get_by_id(self, product_id):
        conn = get_connection()
        result = conn.execute("SELECT * FROM products WHERE id = ?", (product_id,))
        return result.fetchone()

    @cache(ttl=300)
    def get_by_category(self, category_id):
        conn = get_connection()
        result = conn.execute(
            "SELECT * FROM products WHERE category_id = ?", (category_id,)
        )
        return result.fetchall()

    @cache(ttl=600)
    def get_similar(self, product_id):
        conn = get_connection()
        result = conn.execute(
            "SELECT p.* FROM products p WHERE p.category_id = "
            "(SELECT category_id FROM products WHERE id = ?) "
            "ORDER BY p.view_count DESC LIMIT 10", (product_id,)
        )
        return result.fetchall()
```

```
# routes/products.py  (after tactic application)
from repositories.product_repository import ProductRepository

repo = ProductRepository()
```

```python
def get_product(product_id):
    return repo.get_by_id(product_id)

def get_products_by_category(category_id):
    return repo.get_by_category(category_id)
```

The caching change now lives in **2–3 files** (the repository and the cache configuration) instead of 15. Future changes to data access — switching databases, adding query logging, implementing read replicas — similarly require modification in one place. The architectural tactic has reduced the *coupling* between route handlers and the database, *increased the cohesion* of data-access logic, and made the system dramatically more maintainable.

This is exactly the kind of transformation that a well-designed LLM pipeline could automate: detect the scattered database access pattern, select the "Use an Intermediary" tactic, generate the repository class, and rewrite the call sites — all while ensuring the tests still pass.

1.5 Guide Roadmap

This study guide is organized as follows:

Chapter	Title	What you will learn
1	Motivation and Context	Why maintenance is expensive, the knowledge barrier, and the LLM opportunity
2	Software Architecture Foundations	Definitions (Perry & Wolf, Garlan & Shaw), architectural styles, design rationale
3	Quality and Maintainability	ISO/IEC 25010, sub-characteristics (modularity, analysability, modifiability, testability), quality models
4	Architectural Tactics	Tactic taxonomy (Bass et al.), modifiability tactics catalog, tactics vs. patterns
5	Architecture Erosion and Drift	Definitions, symptoms, causes, consequences, detection approaches
6	Maintainability Assessment Methods	Static analysis metrics (CC, MI, Halstead), tools (Radon, SonarQube), tool agreement
7	LLMs for Code Refactoring	Current capabilities, agentic frameworks, multi-agent pipelines, behavior preservation
8	Open Challenges	Inter-procedural synthesis, architecture awareness, evaluation methodology
9	Research Gaps and Thesis Scope	What remains unsolved and how this thesis addresses it

Each chapter builds on the previous ones. By the end, you will have a comprehensive understanding of the theoretical foundations, practical tools, and open research questions surrounding the automated implementation of architectural tactics for software quality improvement.

Chapter 2

Software Architecture Foundations

Learning objectives. After reading this chapter you should be able to (1) state and compare the two foundational definitions of software architecture, (2) explain the role of design rationale as a first-class architectural element, (3) describe the major architectural styles and their quality trade-offs, (4) analyze the KWIC case study as an illustration of style-driven quality differences, and (5) explain how quality attribute scenarios connect styles to architectural tactics.

2.1 What Is Software Architecture?

Before we can discuss *improving* software architecture with automated tactics, we need a precise understanding of what software architecture *is*. Two foundational definitions, published within a year of each other in the early 1990s, established the vocabulary that the entire field still uses today. Each emphasizes different aspects of the same underlying reality, and understanding both provides a richer conceptual foundation.

2.1.1 Perry and Wolf (1992): Elements, Form, and Rationale

Perry and Wolf proposed the first formal model of software architecture in their seminal 1992 paper [7]. Their definition takes the form of a triple:

$$\text{Software Architecture} = \{\text{Elements}, \text{Form}, \text{Rationale}\}$$

Each component of this triple captures a distinct concern.

Elements are the building blocks of an architecture. Perry and Wolf distinguish three classes:

- **Processing elements** perform transformations on data. In a web application, a request handler that validates input and produces a response is a processing element. In a compiler, the lexer, parser, and code generator are each processing elements.

- **Data elements** contain and represent information. The abstract syntax tree passed between compiler phases, the database schema behind a web service, or the message payload in a queue are all data elements.
- **Connecting elements** glue processing and data elements together. A REST API endpoint, a message broker, a shared database, or even a simple function call can serve as a connecting element. Perry and Wolf emphasize that connecting elements are "often overlooked but have a profound impact on the resulting architecture" — systems with identical processing and data elements can have radically different properties depending on how they are connected.

Form comprises two sub-aspects:

- **Properties** constrain which elements may be chosen. For example, a "real-time" property constrains processing elements to those with bounded execution time. Properties are *weighted* — some are mandatory (hard constraints), others desirable (soft constraints).
- **Relationships** constrain how elements may be placed relative to each other. A layered architecture imposes the relationship "layer N may only call layer N-1." A pipes-and-filters architecture imposes "each filter reads from exactly one input pipe and writes to exactly one output pipe."

Rationale captures *why* these elements, properties, and relationships were chosen. It explicates the satisfaction of system constraints including economics, performance, reliability, and organizational factors. Rationale is what distinguishes architecture from mere structure: two systems may have identical elements and form, but different rationale — meaning they were designed to optimize different qualities and should be evolved according to different principles.

A concrete example. Consider a multi-phase compiler:

Element class	Example	Role
Processing	Lexer, Parser, Type Checker, Code Generator	Transform source code through successive representations
Data	Token stream, AST, Symbol table, IR	Intermediate representations passed between phases
Connecting	Function calls (sequential), Shared symbol table (repository)	Link phases together

The *form* might specify: "Each phase must be invocable independently" (property) and "phases execute in a fixed linear sequence" (relationship). The *rationale* explains: "This decomposition was chosen to maximize modifiability — a new optimization pass can be inserted between existing phases without modifying them."

2.1.2 Garlan and Shaw (1993): Components, Connectors, and Configuration

One year later, Garlan and Shaw proposed an alternative formulation that has become equally canonical [10]:

> "Software architecture [is] a collection of computational **components** together with **connectors** describing their interactions, composed into a graph with topological **constraints**."

- **Components** are the computational units that perform the work of the system. They range in granularity from individual objects or modules to entire services or subsystems. A microservice handling user authentication, a database engine, or a JavaScript front-end module are all components. What distinguishes a component from arbitrary code is that it has a well-defined interface through which it communicates with the rest of the system.
- **Connectors** are the interaction mechanisms through which components communicate. Unlike Perry and Wolf's "connecting elements," Garlan and Shaw treat connectors as first-class architectural entities with their own properties and semantics. An HTTP REST call, a message queue, a shared-memory region, a remote procedure call, or a Unix pipe are all connectors. The choice of connector deeply influences the system's quality attributes: a synchronous RPC connector creates tight temporal coupling; an asynchronous message queue provides loose coupling but introduces eventual consistency.
- **Configuration** is the topology — the graph structure describing how components are wired together through connectors. Two systems using the same components and connectors can have very different configurations. A client-server configuration centralizes state in one component; a peer-to-peer configuration distributes it. The configuration determines the system's overall properties: scalability, fault tolerance, modifiability.

A concrete example. Consider a web-based e-commerce system:

Concept	Example	Notes
Component	Product Catalog Service, Order Service, Payment Gateway, PostgreSQL database	Each has a defined API
Connector	REST/HTTP between services, SQL connection to database, Message queue (RabbitMQ) for async order processing	Each has distinct quality implications

Configuration	Services communicate via REST; Order Service publishes events to RabbitMQ; Payment Gateway is called synchronously during checkout	Topology determines coupling and change propagation

2.1.3 Comparing the Two Definitions

Both definitions describe the same phenomenon, but they emphasize different facets. The following table highlights the key differences and commonalities:

Dimension	Perry & Wolf (1992)	Garlan & Shaw (1993)
Core formula	{Elements, Form, Rationale}	Components + Connectors + Configuration
Building blocks	Three element classes (processing, data, connecting)	Two first-class entities (components, connectors)
Data treatment	Data elements are a separate, explicit class	Data is implicit — flows through connectors or resides in components
Connectors	"Connecting elements" — one of three element classes	First-class entities with their own properties; elevated status
Constraints	Form = weighted properties + relationships	Configuration = topological constraints on the component-connector graph
Design motivation	Rationale is a first-class triple member	Not explicitly modeled (implicit in style choice)
Style concept	Architectural style as an abstraction over specific architectures	Styles defined by component/connector types and configuration rules
Emphasis	Formal model; why decisions were made	Taxonomic vocabulary; what structures exist
Origin discipline	Analogy to building, hardware, and network architecture	Emerged from practical SE case studies (KWIC, compilers, etc.)

The two definitions are complementary. Perry and Wolf give us a framework for reasoning about *why* an architecture is the way it is (rationale) and *what constraints* it satisfies (form). Garlan and Shaw give us a concrete vocabulary for describing *what* an architecture looks like (components, connectors, configuration) and *which patterns* recur across systems (styles). A complete understanding of software architecture draws on both.

2.1.4 The "Load-Bearing Walls" Metaphor

Garlan and Perry, writing together in 1995, introduced a metaphor that crystallizes why architecture matters for maintainability [3]:

> "By making explicit the **load-bearing walls** of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications."

In a building, a load-bearing wall supports the structure above it. You can repaint it, hang pictures on it, even change its material — but you cannot remove it without risking collapse. Non-load-bearing partition walls, by contrast, can be moved freely to reconfigure floor plans.

Software architecture works the same way. Some architectural decisions are *load-bearing*:

- The choice to use a relational database (changing to a document store later affects every query in the system)
- The decision to communicate between services via synchronous REST (switching to asynchronous messaging requires rethinking error handling, transaction boundaries, and user experience)
- The decomposition of a monolith into specific service boundaries (merging or splitting services later propagates changes through APIs, deployment pipelines, and team structures)

Other decisions are *partition walls* — easily changed:

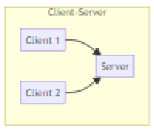
- The specific HTTP framework used within a service (Flask vs. FastAPI can be swapped with modest effort if endpoints are well-isolated)
- The serialization format for internal messages (JSON vs. MessagePack, if abstracted behind a serializer interface)

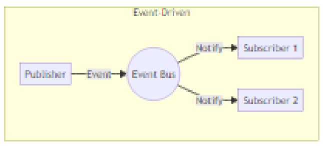
The critical insight for maintainability: **architectural tactics operate on the load-bearing walls.** When we apply "Use an Intermediary" to decouple a database dependency, we are modifying a load-bearing structural decision. When we "Split Module" to separate concerns, we are redefining the structural partitions of the system. These are exactly the changes that are most impactful for long-term maintainability — and exactly the changes that are hardest to get right, which is why automating them is both valuable and challenging.

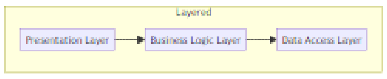
2.2 Architectural Styles

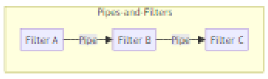
An *architectural style* defines a family of systems in terms of a pattern of structural organization [10]. Each style specifies the types of components and connectors that may be used, how they may be combined, and what constraints govern their composition. Choosing an architectural style is one of the earliest and most consequential decisions in a system's lifecycle, because it determines the system's strengths and weaknesses across multiple quality attributes.

The following diagram illustrates four common architectural styles and their primary structural patterns:









2.2.1 Major Styles Overview

Style	Structure	Quality Strengths	Quality Weaknesses	Example
Pipes-and-Filters	Linear chain of filters connected by pipes; each filter reads input, transforms it, writes output	Reusability (filters are independent), modifiability (add/replace/reorder filters), concurrency (filters can run in parallel)	Poor for interactive systems, overhead from data format conversion between filters, no shared state	Unix shell pipelines (`cat file | grep pattern | sort | uniq`), ETL data pipelines
Layered	Hierarchical layers; each layer provides services to the layer above and consumes services from the layer below	Abstraction (each layer hides complexity), modifiability (change in one layer affects at most adjacent layers), portability (replace a layer implementation)	Performance overhead from indirection, not all systems decompose naturally into layers, "layer bridging" temptation undermines the style	OSI network model, three-tier web applications (presentation / business logic / data)
Event-Driven (Publish-Subscribe)	Components publish events; other components subscribe to event types and react when events occur	Loose coupling (publishers do not know subscribers), extensibility (new subscribers added without changing publishers), evolution (components replaced without interface changes)	Loss of control over processing order, difficulty reasoning about correctness, challenges with data exchange between decoupled components	GUI frameworks (event listeners), microservice event buses, IoT sensor networks

Style	Structure	Quality Strengths	Quality Weaknesses	Example
Repository (Blackboard)	Central data store (repository) accessed by independent processing components; in the Blackboard variant, components are triggered by changes to the shared data	Data integration (single source of truth), tool independence (components interact only through the repository)	Bottleneck at the central store, tight coupling to the data schema, concurrency challenges	Database-centric applications, IDE environments (shared AST), Blackboard AI systems
Client-Server	Clients request services; servers provide them. Clear separation of concerns between request initiation and service provision	Centralized data management, clear security boundaries, shared resource efficiency	Server is a single point of failure, network latency, scalability limited by server capacity	Web applications, email (SMTP/IMAP), databases (client query / server response)
Microservices	System decomposed into small, independently deployable services, each owning its data and communicating via lightweight protocols (typically HTTP/REST or messaging)	Independent deployment, technology heterogeneity, team autonomy, fine-grained scalability	Distributed system complexity (network failures, eventual consistency), operational overhead (monitoring, tracing), data duplication across service boundaries	Netflix, Amazon, modern cloud-native applications

2.2.2 The KWIC Case Study

One of the most influential demonstrations of how architectural style choice impacts quality attributes is the **Key Word In Context (KWIC)** case study, introduced by Parnas in 1972 and elaborated by Garlan and Shaw in 1993 [10].

The problem. A KWIC index system takes a set of lines (e.g., paper titles), generates all circular shifts of each line (moving each word to the front in turn), sorts the shifts alphabetically, and outputs the sorted result. For example, given the title "Software Architecture in Practice," the circular shifts are:

```
Software Architecture in Practice
Architecture in Practice Software
in Practice Software Architecture
Practice Software Architecture in
```

Four architectural decompositions. Garlan and Shaw show how the same KWIC functionality can be implemented using four different styles, each producing a different set of quality trade-offs:

1. **Shared Data (Repository).** All modules access a shared in-memory data structure. The input module stores lines in a shared array; the shift module reads lines and writes shifts to another shared structure; the sort module operates on the shifts in place; the output module reads the sorted shifts.

2. **Abstract Data Types (ADT / Object-Oriented).** Each module encapsulates its own data behind a well-defined interface. The Lines module provides `add_line()`, `get_line()`, `get_char()` operations; the Shifts module provides `shift()`, `get_shift_char()` operations that internally reference the Lines module; and so on.

3. **Implicit Invocation (Event-Driven).** Modules register interest in events. When the Input module finishes reading a line, it publishes a "line-added" event. The Shift module, subscribed to this event, automatically generates shifts. The Sort module is triggered when shifts are complete.

4. **Pipes and Filters.** The system is a pipeline: `Input | CircularShift | Sort | Output`. Each stage reads from standard input, processes, and writes to standard output. Filters are independent processes with no shared state.

Quality trade-off comparison. The following table, adapted from Garlan and Shaw's analysis, compares the four decompositions across five quality dimensions:

Quality dimension	Shared Data	ADT / OO	Implicit Invocation	Pipes and Filters
Algorithm change (e.g., replace sort algorithm)	Poor — sort is tangled with shared data structure	Good — encapsulated behind interface	Good — replace subscriber	Best — replace a filter

Quality dimension	Shared Data	ADT / OO	Implicit Invocation	Pipes and Filters
Data representation change (e.g., change from array to linked list)	Poor — all modules depend on shared structure	Best — hidden behind ADT interface	Good — data is local to components	Good — filter-internal data is private
Functional enhancement (e.g., add "stop words" filtering)	Poor — requires modifying shared data structures	Good — add a new ADT module	Best — add a new subscriber	Best — insert a new filter
Performance (minimize memory, maximize speed)	Best — direct memory access, no overhead	Good — some overhead from interface calls	Poor — event dispatch overhead	Poor — data copying between filters
Reusability (use components in other systems)	Poor — tightly coupled to shared data	Good — modules are self-contained	Good — event-based components are portable	Best — filters are fully independent

Key insight. No single style dominates all dimensions. The Shared Data style wins on performance but loses on modifiability. Pipes and Filters excels at reusability and functional enhancement but pays a performance cost. The ADT style balances most dimensions but does not achieve the best score in any single one. This is why architectural design involves *trade-offs* — and why the concept of *architectural tactics* (fine-grained decisions targeting specific qualities) is essential.

2.2.3 Heterogeneous Architectures

Real-world systems rarely conform to a single pure architectural style. Garlan and Shaw explicitly note that practical architectures are **heterogeneous**, combining multiple styles through several mechanisms [10]:

- **Hierarchical composition.** A system may be organized as a layered architecture at the top level, but within a specific layer, use a pipes-and-filters decomposition. For example, a data processing layer might internally use an ETL pipeline, while the overall system follows a client-server pattern.
- **Mixed connectors.** A single system may use REST calls between services, message queues for asynchronous operations, and shared databases for batch processing — combining client-server, event-driven, and repository connectors.
- **Multi-level elaboration.** A high-level "microservices" style decomposes into individual services, each of which internally follows a layered (controller / service / repository) pattern.

This heterogeneity is important for the thesis because it means that automated tactic implementation cannot assume a single style. An LLM-based tool must recognize the local style context of the code it is modifying and select tactics that are compatible with that context. As Harrison and Avgeriou demonstrated, the same tactic can be a "Good Fit" in one style and a "Poor Fit" in another [11].

2.3 Design Rationale

2.3.1 Rationale as a First-Class Element

Perry and Wolf's inclusion of *rationale* in the architecture triple is one of their most important and most overlooked contributions [7]. Rationale answers the question: **Why was this architecture chosen?**

Consider two systems with identical structure: both use a layered architecture with three tiers. But the rationale differs:

- **System A** was layered to maximize *portability*: the data layer abstracts the database engine so the system can run on PostgreSQL, MySQL, or SQLite.
- **System B** was layered to maximize *team autonomy*: the presentation layer is owned by the front-end team, the business layer by the back-end team, and the data layer by the DBA team.

These systems look the same in a class diagram, but they should be *evolved differently*. In System A, changes to the data layer interface are extremely costly (they compromise portability). In System B, changes to the data layer interface may be acceptable if they improve the back-end team's velocity, as long as the inter-team API contract is maintained.

Without rationale, a maintainer looking at either system sees only the structure and might make changes that violate the original design intent — contributing to architectural erosion.

2.3.2 The Documentation Problem

In practice, design rationale is rarely documented and even more rarely traced from architecture decisions to code [3], [7]. This creates a persistent problem:

1. The original architects understand *why* the system is structured as it is.
2. As team members rotate, this knowledge dissipates — Perry and Wolf's concept of **knowledge vaporization** [7]. Li et al. confirmed this as a top non-technical cause of architecture erosion, cited in 15.1% of the studies they surveyed [4].
3. New developers, lacking rationale, make changes that seem locally reasonable but violate architectural constraints.
4. The architecture erodes. Erosion increases the cost of future changes. The maintenance crisis deepens.

This documentation gap is part of what makes automated tactic implementation so valuable: if the tactic catalog explicitly encodes design rationale (e.g., "Use an Intermediary *because* direct coupling between modules increases change propagation cost"), then the

LLM pipeline preserves and propagates that rationale through the code changes it generates.

2.3.3 Connecting Rationale to Tactics

Architectural tactics are, in essence, the **operational expression of rationale**. When a quality attribute requirement states "the system must accommodate new data sources without modifying existing processing modules," the rationale for choosing the "Generalize Module" tactic is directly embedded in that requirement. The relationship is:

```
Quality Attribute Requirement (WHY)
    -> Tactic Selection (WHAT design decision)
        -> Code Transformation (HOW it is implemented)
```

Perry and Wolf's model tells us that the *rationale* motivates the *form* (properties and relationships), and the form constrains the *elements*. Tactics are the mechanism by which we translate rationale into form: each tactic imposes specific properties ("modules must be decoupled") and relationships ("communication must go through an intermediary") that shape the architecture.

2.4 From Styles to Tactics

2.4.1 The Granularity Gap

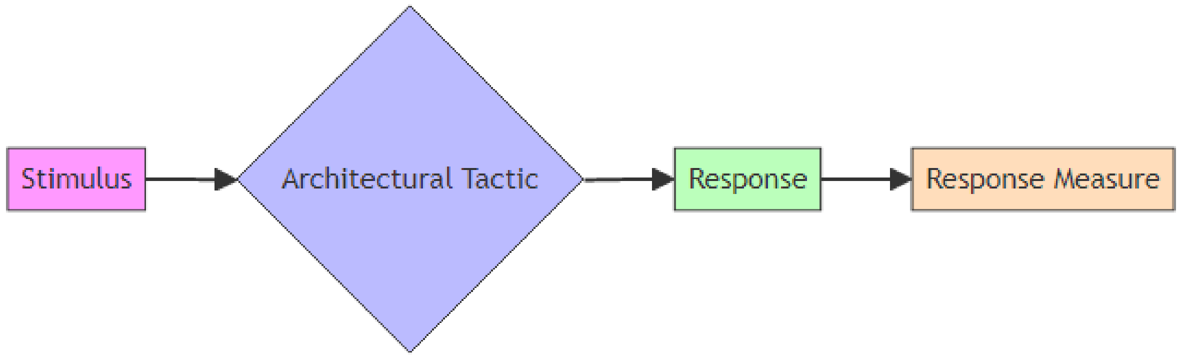
Architectural styles provide the *big picture*: the overall structural organization of a system. But styles alone do not prescribe how to handle specific quality attribute requirements. Knowing that a system uses a layered architecture does not tell you:

- How to ensure that a change to the payment processing logic does not cascade into the presentation layer.
- How to reduce the coupling between the authentication module and every service that requires authorization.
- How to make the logging infrastructure replaceable at deployment time.

These are finer-grained design decisions that operate *within* a chosen style. They are **architectural tactics** — a concept introduced and systematically cataloged by Bass, Clements, and Kazman [1].

2.4.2 Quality Attribute Scenarios

The following diagram shows how quality attribute scenarios connect styles, tactics, and code:



Bass et al. define a formal structure for reasoning about quality requirements called a **quality attribute scenario** [1]. Each scenario has six parts:

Part	Description	Example
Source of stimulus	An entity (human, system, environment) that generates the stimulus	A developer on the maintenance team
Stimulus	A condition that the architecture must respond to	Wants to add support for a new payment provider
Environment	The conditions under which the stimulus occurs	During normal development, system is in production
Artifact	The part of the system that is stimulated	The payment processing module
Response	The activity that results from the stimulus	The change is made, tested, and deployed
Response measure	How the response is measured	Change is confined to 1 module, completed in < 4 hours, no regression failures

A modifiability scenario might read: "A developer (source) wants to add a new payment provider (stimulus) during normal development (environment). The change should affect only the payment module (artifact), be completed within 4 person-hours (response measure), and introduce no regressions (response measure)."

This scenario *motivates* the selection of specific tactics. To confine the change to one module, we might apply "Use Encapsulation" (hide the payment provider behind a stable interface) and "Generalize Module" (make the payment module parameterized by provider). To avoid regressions, we might apply "Maintain Existing Interface" (ensure the new provider conforms to the existing contract).

2.4.3 Bridging to the Next Chapter

Quality attribute scenarios are the bridge between architectural styles (Chapter 2) and architectural tactics (Chapter 4). Styles determine the overall playing field; scenarios define the specific quality goals; and tactics are the design moves that achieve those goals within the constraints of the chosen style.

In the next chapter, we examine the quality model that gives us a precise vocabulary for *what* we mean by "maintainability" — the ISO/IEC 25010 standard and its sub-characteristics. In Chapter 4, we then present the complete catalog of modifiability tactics and show how each one addresses specific maintainability concerns.

Review questions.

1. In Perry and Wolf's model, what is the difference between *form* and *rationale*? Why is rationale important for long-term maintenance?
2. Using Garlan and Shaw's vocabulary, describe the components, connectors, and configuration of a system you have worked on or studied.
3. In the KWIC case study, why does the Pipes-and-Filters style excel at functional enhancement but perform poorly on raw execution speed?
4. Give an example of a "load-bearing" architectural decision and a "partition wall" decision in a web application you are familiar with.
5. How does the concept of *design rationale* connect to the problem of *architecture erosion*?
6. Write a quality attribute scenario (source, stimulus, environment, artifact, response, response measure) for a modifiability requirement in a system you know.

Chapter 3

Software Quality & Maintainability

Software quality is not a monolithic concept. Over five decades of software engineering research, the community has progressively refined how we define, decompose, and measure quality – with maintainability emerging as one of the most economically significant and technically challenging attributes. This chapter traces that evolution, establishes ISO/IEC 25010 as the authoritative framework, and shows how maintainability can be operationalized through concrete metrics and guidelines.

3.1 Evolution of Quality Models

Understanding modern maintainability requires understanding the intellectual lineage that produced it. Four major quality models, spanning from 1977 to 2011, have shaped how we think about software quality attributes. Each generation addressed limitations of its predecessors while introducing new perspectives and decompositions.

3.1.1 McCall's Quality Model (1977)

McCall's model was the first systematic attempt to define and measure software quality. Developed for the U.S. Air Force, it introduced 11 quality factors organized into three perspectives based on how the software is used [12]:

- **Product Operation** (daily use): correctness, reliability, efficiency, integrity, usability
- **Product Revision** (change and improvement): maintainability, flexibility, testability
- **Product Transition** (adaptation to new environments): portability, reusability, interoperability

McCall's key innovation was a hierarchical structure: **factors** (user-oriented) decompose into **criteria** (developer-oriented), which decompose into **metrics** (measurable indicators). For example, the factor *maintainability* decomposed into criteria like consistency, simplicity, conciseness, self-descriptiveness, and modularity.

However, McCall's metrics were "neither clearly nor completely defined" [12], making practical measurement difficult. The model also treated quality factors as relatively independent, failing to capture the tradeoffs that architects face in practice (e.g., improving efficiency often reduces maintainability).

3.1.2 Boehm's Quality Model (1978)

Boehm proposed a hierarchical model that began with a top-level question: "Does the software do what the user wants it to do?" This led to a tree structure rooted in **general utility**, which branched into three primary concerns:

- **As-is utility** – the software works correctly and efficiently now
- **Maintainability** – the software can be understood, modified, and tested
- **Portability** – the software can be moved to new environments

Boehm's model contributed two important ideas. First, it placed maintainability as a first-class top-level concern rather than burying it among many peer factors. Second, it introduced the notion that quality attributes have a natural hierarchy where some attributes serve others. Maintainability in Boehm's model decomposed into understandability, modifiability, and testability – a decomposition remarkably close to what ISO 25010 would formalize three decades later.

The limitation of Boehm's model was its focus on high-level characteristics without providing operational metrics or thresholds. It told architects *what* to care about but not *how* to measure it [13].

3.1.3 ISO 9126 (2001)

ISO 9126 represented the first international consensus on software quality. Developed by the International Organization for Standardization, it defined six quality characteristics, each with sub-characteristics:

1. **Functionality** – suitability, accuracy, interoperability, security, compliance
2. **Reliability** – maturity, fault tolerance, recoverability, compliance
3. **Usability** – understandability, learnability, operability, attractiveness, compliance
4. **Efficiency** – time behavior, resource utilization, compliance
5. **Maintainability** – analyzability, changeability, stability, testability, compliance
6. **Portability** – adaptability, installability, co-existence, replaceability, compliance

ISO 9126 decomposed maintainability into four sub-characteristics:

- **Analyzability:** the effort needed to diagnose deficiencies or identify parts to be modified
- **Changeability:** the effort needed to implement a specified modification
- **Stability:** the risk of unexpected effects from modifications
- **Testability:** the effort needed to validate modified software

This model consolidated decades of quality research into a standardized vocabulary. Comparative analyses demonstrate that ISO 9126 provided the most balanced factor coverage among pre-2010 models [12], [13]. Al-Badareen et al. quantified this using a weight-based comparison method, finding that McCall scored 63.67% in overall factor coverage while FURPS (a simpler model from Hewlett-Packard) scored only 20.34% – with ISO 9126 providing the most standardized and complete framework [13].

3.1.4 ISO/IEC 25010 (2011)

ISO/IEC 25010 refined its predecessor by expanding to eight quality characteristics and restructuring sub-characteristics based on a decade of industry experience [14]. The key changes relevant to maintainability were:

- "Changeability" was renamed to **modifiability** (a broader, more precise term)
- **Modularity** and **reusability** were added as explicit sub-characteristics
- "Stability" was subsumed by modifiability (modification without degradation)
- Security was elevated from a sub-characteristic of functionality to a top-level characteristic

3.1.5 Evolution Comparison Table

The following table traces how the concept of "ease of change" evolved across quality models:

Aspect	McCall (1977)	Boehm (1978)	ISO 9126 (2001)	ISO/IEC 25010 (2011)
Top-level structure	3 perspectives, 11 factors	Hierarchical utility tree	6 characteristics	8 characteristics
Maintainability level	One of 11 factors	Top-level branch (1 of 3)	One of 6 characteristics	One of 8 characteristics
"Ease of change" term	Flexibility	Modifiability	Changeability	Modifiability
Decomposition depth	Factors > Criteria > Metrics	Characteristics > Primitives	Characteristics > Sub-characteristics	Characteristics > Sub-characteristics
Sub-characteristics	Simplicity, conciseness, self-descriptiveness, modularity, consistency	Understandability, modifiability, testability	Analyzability, changeability, stability, testability	**Modularity, reusability, analysability, modifiability, testability**
Metric operationalization	Poorly defined	Absent	Partially defined (external metrics)	Quality-in-use + product quality metrics
International consensus	No (US Air Force)	No (academic)	Yes (ISO)	Yes (ISO/IEC JTC1)
Coverage score	63.67%	N/A	Highest standardized	Supersedes ISO 9126

The progression shows a clear trend: maintainability has been refined from a vaguely defined “ability to fix bugs” into a multi-dimensional attribute with five distinct, measurable sub-characteristics. This thesis adopts ISO/IEC 25010 as its evaluation framework because it represents the current international consensus and provides the most fine-grained decomposition of maintainability available.

3.2 ISO/IEC 25010: The Standard Framework

ISO/IEC 25010 defines two quality models: a **product quality model** (8 characteristics describing internal and external quality) and a **quality-in-use model** (5 characteristics describing the user’s experience). For evaluating architectural changes, the product quality model is the relevant one.

3.2.1 The Eight Product Quality Characteristics

#	Characteristic	Description
1	**Functional Suitability**	Degree to which a product provides functions that meet stated and implied needs
2	**Performance Efficiency**	Performance relative to the amount of resources used
3	**Compatibility**	Degree to which a product can exchange information with and perform functions alongside other products
4	**Usability**	Degree to which a product can be used to achieve goals with effectiveness, efficiency, and satisfaction
5	**Reliability**	Degree to which a system performs specified functions under specified conditions for a specified period
6	**Security**	Degree to which a product protects information and data
7	**Maintainability**	Degree of effectiveness and efficiency with which a product can be modified
8	**Portability**	Degree to which a system can be transferred from one environment to another

3.2.2 Deep Dive: Maintainability and Its Five Sub-Characteristics

Maintainability in ISO/IEC 25010 is decomposed into five sub-characteristics that together capture the full spectrum of what it means for software to be “easy to change.” Each sub-characteristic targets a different aspect of the modification lifecycle – from understanding what needs to change, to making the change, to verifying it worked correctly.

Sub-characteristic	ISO Definition	Code-Level Indicators	Architecture-Level Indicators
Modularity	Degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components	Low coupling between classes; high cohesion within classes; small, focused modules; absence of God classes	Clear component boundaries; well-defined interfaces; acyclic dependency graph; balanced component sizes
Reusability	Degree to which an asset can be used in more than one system, or in building other assets	Generic utility functions; parameterized classes; abstract base classes; absence of hardcoded dependencies	Shared service layers; library extraction; protocol-based interfaces; standardized data formats
Analysability	Degree of effectiveness and efficiency with which it is possible to assess the impact of an intended change, to diagnose deficiencies or causes of failures, or to identify parts to be modified	Clear naming conventions; comprehensive documentation; low cyclomatic complexity; small method size	Traceable architectural decisions; consistent layering; observable system state; separation of cross-cutting concerns

Sub-characteristic	ISO Definition	Code-Level Indicators	Architecture-Level Indicators
Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality	Low coupling between objects (CBO); small method parameter lists; absence of feature envy; encapsulated data	Loose coupling between components; intermediary layers; stable interfaces; deferred binding strategies
Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system and tests can be performed to determine whether those criteria have been met	Dependency injection; small testable units; deterministic behavior; absence of global state	Clear input/output contracts; mockable interfaces; observable side effects; isolated components

Understanding these sub-characteristics is essential because architectural tactics target specific sub-characteristics. For example, the "Split Module" tactic primarily targets **modularity** (creating smaller, more independent components) and **modifiability** (localizing the impact of changes). When the thesis evaluates whether an LLM-implemented tactic succeeded, it must measure improvement in the targeted sub-characteristic specifically, not just a generic "maintainability score."

3.2.3 Quality Attribute Scenarios

Bass, Clements, and Kazman provide a structured method for specifying quality requirements called **quality attribute scenarios** [1]. A scenario has six parts:

1. **Source of stimulus** – Who or what causes the change?
2. **Stimulus** – What change or event occurs?
3. **Environment** – Under what conditions? (design time, build time, runtime)
4. **Artifact** – What part of the system is affected?
5. **Response** – What should happen?

6. **Response measure** – How do we know it succeeded?

A concrete maintainability scenario might read:

> A developer (source) wants to replace the payment gateway library (stimulus) during development (environment). The payment processing module (artifact) is modified with no side effects on other modules (response). The change is completed in fewer than 3 person-hours, affecting at most 2 modules, with all existing tests passing (response measure).

Quality attribute scenarios bridge the gap between abstract quality definitions and concrete, testable requirements. They provide the "before" and "after" framing that makes maintainability measurable.

3.3 Maintainability in Practice

3.3.1 The 60-80% Problem

The economic importance of maintainability is difficult to overstate. Industry-wide data consistently shows that maintenance activities consume 60-80% of the total software lifecycle cost [1]. For a typical enterprise system with a 15-year lifespan, this means that for every dollar spent on initial development, three to four dollars are spent on maintenance activities – understanding code, diagnosing defects, implementing changes, and verifying correctness.

This cost distribution has a profound architectural implication: design decisions that reduce maintenance effort, even by small percentages, compound over the system's lifetime into substantial savings. A 10% reduction in the time needed to understand and modify a module translates to thousands of person-hours saved across a large codebase over a decade. This economic reality is what motivates the study of architectural tactics for maintainability improvement.

3.3.2 Empirical Evidence: Molnar's Longitudinal Study

Molnar and Motogna provide the most rigorous longitudinal evidence for how maintainability behaves in real, evolving software systems [15]. Their study analyzed 111 releases of three open-source Java applications (FreeMind, jEdit, TuxGuitar), each spanning over a decade of development, using three quantitative maintainability models:

1. **Maintainability Index (MI):** A composite formula combining Halstead volume, cyclomatic complexity, lines of code, and optionally comment percentage. Widely used but has known limitations.

2. **ARiSA Compendium Model:** An object-oriented metrics model that measures maintainability through class-level metrics including coupling, cohesion, complexity, and size.

3. **SQALE (Software Quality Assessment based on Lifecycle Expectations):** A technical debt model implemented in SonarQube that estimates the effort required to fix all identified code issues, expressed as a ratio of remediation effort to development effort.

Their key findings reshape how we should think about measuring maintainability:

Finding	Implication for Practice
SQALE is the most reliable system-level model	Use SonarQube Technical Debt Ratio for before/after comparisons at the system level
MI is confounded by system size at the system level	Do not rely on MI alone for cross-project or cross-version comparisons
MI correlates with SQALE at the class level (Spearman rho ~ -0.6)	MI remains useful for quick, fine-grained method/class complexity screening
Maintainability effort concentrates in a small subset of packages	Target "hotspot" packages for maximum impact – in jEdit, 6 packages account for ~80% of maintenance effort
Mature application versions stabilize in quality	Early architectural investment pays compounding dividends over time
Major feature additions cause maintainability spikes	Milestone releases (e.g., FreeMind 0.8.0) can dramatically decrease maintainability if not accompanied by deliberate refactoring
Auto-generated code can dominate metrics	A single auto-generated package in FreeMind was the primary driver of its worst maintainability scores

The hotspot concentration finding is particularly important for the thesis approach. If 80% of maintenance effort is concentrated in a small number of modules, then an LLM-based tactic implementation pipeline should prioritize identifying and transforming those specific modules rather than applying tactics uniformly across the entire codebase.

3.3.3 Practical Example: Analyzing a Python Module with Radon

To make maintainability measurement concrete, consider a Python module analyzed with Radon, a static analysis tool that computes cyclomatic complexity (CC) and maintainability index (MI).

Given the following module:

```python
# payment_processor.py - A module with multiple responsibilities
import json
import logging
from datetime import datetime

class PaymentProcessor:
    def __init__(self, config_path):
        with open(config_path) as f:
            self.config = json.load(f)
        self.logger = logging.getLogger(__name__)
        self.transactions = []

    def process_payment(self, amount, currency, card_number,
                        cvv, expiry, customer_id, order_id,
                        billing_address, shipping_address):
        # Validate card
        if len(card_number) != 16:
            raise ValueError("Invalid card number")
        if len(cvv) != 3:
            raise ValueError("Invalid CVV")
        month, year = expiry.split("/")
        if int(year) < datetime.now().year % 100:
            raise ValueError("Card expired")
        elif int(year) == datetime.now().year % 100:
            if int(month) < datetime.now().month:
                raise ValueError("Card expired")

        # Calculate fees
        if currency == "USD":
            fee = amount * 0.029 + 0.30
        elif currency == "EUR":
            fee = amount * 0.034 + 0.25
        elif currency == "GBP":
            fee = amount * 0.034 + 0.20
        else:
            fee = amount * 0.045 + 0.50

        # Process
        total = amount + fee
        transaction = {
            "id": f"TXN-{order_id}-{datetime.now().timestamp()}",
            "amount": amount,
            "fee": fee,
            "total": total,
            "currency": currency,
            "customer_id": customer_id,
            "order_id": order_id,
            "status": "completed",
            "timestamp": datetime.now().isoformat()
        }
        self.transactions.append(transaction)

        # Log
        self.logger.info(f"Payment processed: {transaction['id']}")

        # Generate receipt
        receipt = f"""
        ====== RECEIPT ======
        Transaction: {transaction['id']}
        Amount: {amount} {currency}
        Fee: {fee} {currency}
        Total: {total} {currency}
        Date: {transaction['timestamp']}
        =====================
        """
        return transaction, receipt

    def get_daily_report(self, date):
        daily = [t for t in self.transactions
                 if t["timestamp"].startswith(date)]
        total_amount = sum(t["amount"] for t in daily)
        total_fees = sum(t["fee"] for t in daily)
        if len(daily) == 0:
            avg = 0
        else:
            avg = total_amount / len(daily)
        return {
            "date": date,
            "count": len(daily),
            "total_amount": total_amount,
            "total_fees": total_fees,
            "average": avg
        }
```

Running Radon on this module:

```
$ radon cc payment_processor.py -s -a
payment_processor.py
    C 7:0 PaymentProcessor
        M 8:4 __init__ - A (1)
        M 13:4 process_payment - C (12)
        M 57:4 get_daily_report - B (6)

Average complexity: B (6.33)

$ radon mi payment_processor.py -s
payment_processor.py - B (38.21)
```

Interpreting the results:

- **Cyclomatic Complexity (CC):** The `process_payment` method scores **C (12)**, meaning it has 12 independent paths through the code. Radon grades CC as: A (1-5, low risk), B (6-10, moderate), C (11-15, high risk), D (16-20, very high risk), E/F (>20, untestable). A CC of 12 indicates this method is difficult to test thoroughly and likely to harbor bugs in edge cases.
- **Maintainability Index (MI):** The module scores **B (38.21)** on a scale where A (>20) is maintainable, B (10-20) is moderately maintainable, and C (<10) is

difficult to maintain. Note: Radon uses a 0-100 scale where higher is better. A score of 38.21 is above the threshold but signals room for improvement.

The problems are identifiable from the metrics: - `process_payment` has **9 parameters** (violating Visser's Guideline 4: keep unit interfaces small, threshold <= 4) - The method handles validation, fee calculation, transaction recording, logging, AND receipt generation (violating the Single Responsibility Principle) - Multiple responsibility areas are interleaved in a single method body, reducing analysability

This is precisely the kind of module where architectural tactics – specifically *Split Module* and *Increase Semantic Coherence* – would improve measurable maintainability.

3.4 The SIG Maintainability Model

The Software Improvement Group (SIG), based on the work of Joost Visser and colleagues, developed a practitioner-oriented maintainability model that bridges the gap between abstract quality standards and daily coding decisions [16]. The model is grounded in the analysis of hundreds of real-world software systems and provides benchmarked thresholds for maintainability metrics.

3.4.1 The 10 Guidelines

Visser's model defines 10 guidelines for writing maintainable code, each tied to a specific, measurable metric:

#	Guideline	Metric	Threshold	Rationale
1	**Write Short Units of Code**	Lines of code per unit (method/function)	<= 15 lines	Short units are easier to understand, test, and reuse. Long methods tend to accumulate multiple responsibilities.
2	**Write Simple Units of Code**	McCabe Cyclomatic Complexity per unit	<= 5	Simple control flow reduces the number of test cases needed and makes behavior predictable. Each branch point adds a potential source of errors.

#	Guideline	Metric	Threshold	Rationale
3	**Write Code Once (DRY)**	Code duplication percentage	Low % (minimize)	Duplicated code means duplicated bugs and duplicated maintenance effort. A fix in one copy must be replicated in all copies – a process that is error-prone and often incomplete.
4	**Keep Unit Interfaces Small**	Number of parameters per method/function	<= 4	Large parameter lists indicate that a method is doing too much or that related parameters should be grouped into objects. They also make method calls harder to read and more error-prone.
5	**Separate Concerns in Modules**	Module coupling (fan-in)	Low coupling between modules	Each module should have a single, well-defined responsibility. When concerns are separated, changes to one concern do not ripple through unrelated modules.
6	**Couple Architecture Components Loosely**	Component independence (% of hidden/internal code)	High % hidden	Components that expose minimal public interfaces are easier to replace, evolve, and test independently. The percentage of code that is "hidden" (not part of the public API) indicates encapsulation quality.

#	Guideline	Metric	Threshold	Rationale
7	**Keep Architecture Components Balanced**	Component count (6-12 ideal) + Gini coefficient	Balanced distribution	A system with one massive component and many tiny ones is poorly decomposed. Balanced components indicate well-distributed responsibilities. The Gini coefficient measures inequality in component sizes.
8	**Keep Your Codebase Small**	Total lines of code / rebuild value	Minimize	Larger codebases require more effort to understand, navigate, and maintain. Every line of code is a liability that must be read, understood, and potentially modified.
9	**Automate Tests**	Test coverage percentage	>= 80%	Automated tests provide a safety net for modifications. Without sufficient coverage, developers cannot confidently change code because they have no way to verify that existing behavior is preserved.

10	**Write Clean Code**	Number of code smells / static analysis findings	Minimize	Code smells (long methods, duplicated code, dead code, commented-out code) reduce readability and signal structural problems. Clean code communicates intent clearly.

3.4.2 The Star Rating System

The SIG model uses a 1-5 star rating system calibrated against a benchmark of hundreds of real-world systems:

Stars	Percentile	Interpretation
5 stars	Top 5%	Best in class – the system is among the most maintainable in the benchmark
4 stars	Top 30%	Above average – the system meets industry standards for maintainability
3 stars	Average (30-70%)	Average – typical for the industry but with clear improvement opportunities
2 stars	Bottom 30%	Below average – maintainability issues are likely causing visible productivity problems
1 star	Bottom 5%	Worst in class – the system likely has severe maintainability problems

The practical significance of these ratings is backed by empirical data. SIG's benchmarking found that issue resolution is approximately **2x faster in 4-star systems compared to 2-star systems** [16]. This means that a team working on a well-maintained codebase can fix bugs and deliver features in roughly half the time of a team working on a poorly maintained one.

3.4.3 Mapping Guidelines to ISO 25010 and Metrics

Each of Visser's guidelines contributes to one or more ISO 25010 maintainability sub-characteristics. Understanding this mapping is essential for connecting practical coding guidelines to the formal quality framework:

#	Guideline	Primary ISO 25010 Sub-characteristic	Secondary Sub-characteristic	Measurable Metric	Tool
1	Write Short Units	Analysability	Modifiability	Unit LOC	Radon (raw metrics)
2	Write Simple Units	Analysability, Testability	Modifiability	Cyclomatic Complexity	Radon (`radon cc`)
3	Write Code Once (DRY)	Modifiability	Reusability	Duplication %	SonarQube, PMD/CPD
4	Keep Unit Interfaces Small	Modifiability	Reusability	Parameter count	Radon, pylint
5	Separate Concerns	Modularity	Analysability	Module fan-in/fan-out	pydeps, import-linter
6	Loose Coupling	Modularity	Testability	Hidden code %, CBO	SonarQube, custom
7	Balanced Components	Modularity	Analysability	Component Gini coefficient	Custom analysis
8	Small Codebase	Analysability	All	Total LOC	cloc, Radon
9	Automate Tests	Testability	Modifiability	Coverage %	coverage.py, pytest-cov
10	Write Clean Code	Analysability	Modifiability	Code smell count	SonarQube, flake8, pylint

3.4.4 Connecting SIG Guidelines to Architectural Tactics

A key insight, noted by Visser and confirmed by mapping his guidelines to the Bass taxonomy, is that **unit-level code quality aggregates upward to determine architecture-level maintainability** [16]. The 10 guidelines map directly onto the three categories of modifiability tactics defined by Bass et al. [1]:

Visser Guideline(s)	Bass Tactic Category	Specific Tactics
Short Units (1), Simple Units (2), Separate Concerns (5)	**Increase Cohesion**	Split Module, Increase Semantic Coherence

Visser Guideline(s)	Bass Tactic Category	Specific Tactics
Loose Coupling at module level (5) and architecture level (6)	**Reduce Coupling**	Restrict Dependencies, Use Encapsulation
Write Code Once (3), Small Interfaces (4)	**Abstract Common Functionality**	Abstract Common Services, Generalize Module
Write Clean Code (10) – incremental cleanup	Evolutionary maintenance	Boy Scout Rule (continuous small improvements)

This mapping demonstrates that the gap between "good coding practices" and "architectural tactics" is smaller than it might appear. Many architectural tactics are, at their core, disciplined applications of coding guidelines at the module and system level. This observation is important for the thesis because it suggests that LLMs, which have demonstrated competence at code-level refactoring tasks, may be able to implement architectural tactics if given appropriate context about the system's structure and the tactic's intent.

3.4.5 A Critical Caveat

It is important to note that while the SIG model provides excellent operationalization of maintainability, its benchmark thresholds are proprietary and recalibrated yearly. For the thesis, we use open-source equivalents (Radon for complexity metrics, SonarQube Community Edition for technical debt analysis, coverage.py for test coverage) with transparent thresholds. The SIG thresholds serve as reference guidelines, not as absolute standards. What matters for evaluating LLM-implemented tactics is the **direction and magnitude of metric change**, not whether a specific threshold is crossed.

3.5 Summary

This chapter established the theoretical and practical foundation for measuring maintainability. The key takeaways are:

1. **Quality models have converged** on ISO/IEC 25010, which decomposes maintainability into five measurable sub-characteristics: modularity, reusability, analysability, modifiability, and testability.

2. **Maintainability dominates lifecycle costs** (60-80%), making even small improvements economically significant, especially when compounded over a system's lifespan.

3. **SQALE/Technical Debt Ratio** is the most reliable system-level maintainability metric, while the Maintainability Index remains useful at the class/method level for fine-grained assessment [15].

4. **Maintenance effort concentrates in hotspots** – a small fraction of modules contains the majority of maintainability issues, suggesting that targeted architectural interventions can yield disproportionate improvements.

5. **The SIG model's 10 guidelines** provide concrete, metric-backed coding practices that map directly to the modifiability tactics discussed in the next chapter [16].
6. **The connection between code-level metrics and architectural tactics** creates a measurable bridge: tactics define *what* architectural change to make, metrics define *how* to evaluate whether it improved maintainability.

The next chapter introduces architectural tactics in detail – the specific design decisions that, when applied systematically, target and improve these measurable quality attributes.

Chapter 4

Architectural Tactics

Architectural tactics are the core mechanism through which architects translate quality attribute requirements into concrete design decisions. While the previous chapter established *what* maintainability means and *how* to measure it, this chapter addresses the more fundamental question: *what specific design decisions improve maintainability, and how can they be systematically applied?* This is the central chapter of the study guide because the thesis proposes automating the implementation of these tactics using large language models.

4.1 What Are Architectural Tactics?

4.1.1 Definition

An architectural tactic is formally defined as:

> "A design decision that influences the achievement of a quality attribute response." [1]

More concretely, a tactic is a targeted architectural intervention that addresses a single quality attribute concern. If a system needs to be modifiable, there are specific, named design decisions – like encapsulating implementation behind interfaces or splitting large modules into focused components – that an architect can apply. These named decisions are tactics.

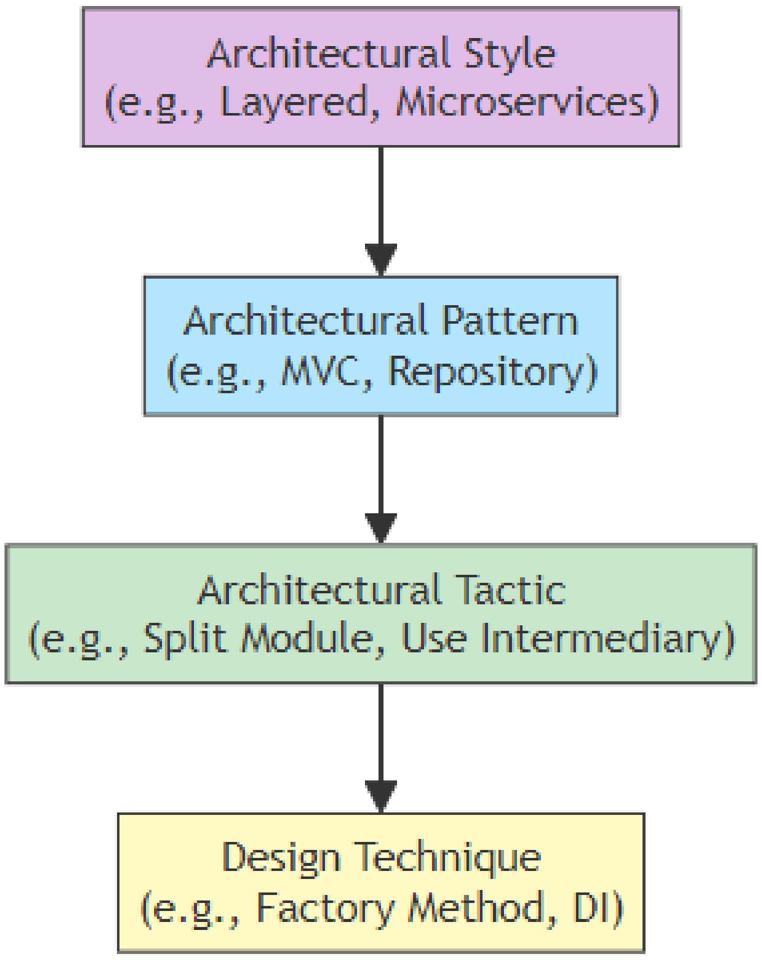
Kim et al. elaborate on this definition:

> "An architectural tactic is a fine-grained reusable architectural building block that provides an architectural solution built from experience to help to achieve a quality attribute." [17]

The word "fine-grained" is critical. Tactics are deliberately small and focused. Each tactic targets a single quality attribute response, without worrying about side effects on other quality attributes. This is what distinguishes them from patterns, which are larger, more complex structures that inherently embed tradeoff decisions.

4.1.2 The Design Hierarchy: Style, Pattern, Tactic, Technique

To understand tactics properly, you must understand where they sit in the hierarchy of architectural design concepts:



From most abstract to most concrete:

Level	Concept	Scope	Example	Quality Scope
Style	Overall structural organization	System-wide	Layered, Microservices, Event-Driven, Pipe-and-Filter	Multiple quality attributes simultaneously
Pattern	Recurring solution template	Subsystem or component group	MVC, Repository, Observer, Broker	Multiple quality attributes with built-in tradeoffs
Tactic	Targeted quality attribute decision	Component or interface level	Split Module, Use an Intermediary, Publish-Subscribe	**Single quality attribute**

Level	Concept	Scope	Example	Quality Scope
Design Technique	Code-level implementation	Class or method level	Factory Method, Dependency Injection, Strategy Pattern	Implementation of a tactic

Bass et al. explain the relationship between tactics and patterns:

> "Architectural tactics are 'building blocks' from which architecture patterns are created; patterns package tactics." [1]

This means that when you look inside any architectural pattern, you find a collection of tactics working together. The MVC pattern, for example, packages the "Split Module" tactic (separating model, view, and controller into distinct components), the "Use Encapsulation" tactic (each component hides its internals), and the "Restrict Dependencies" tactic (the view cannot directly modify the model without going through the controller). Understanding this decomposition is what allows us to reason about *which specific aspect* of a pattern produces *which specific quality attribute benefit*.

4.1.3 The Quality Attribute Scenario Model

Tactics do not exist in a vacuum. They are applied in response to specific quality attribute scenarios. Bass et al. define a six-part scenario model that structures how we think about quality requirements and the tactics that address them [1]:

Part	Description	Example (Modifiability)
Source of stimulus	The entity that causes the change	A developer on the team
Stimulus	The change or event that must be accommodated	Wants to replace the payment gateway from Stripe to PayPal
Environment	The conditions under which the stimulus occurs	Design time (the system is not running)
Artifact	The part of the system affected	The payment processing module
Response	What the system (or development process) does	The change is made with no side effects on other modules
Response measure	How we know the response was satisfactory	Completed in less than 3 person-hours, affecting at most 2 modules, all tests pass

The tactic is the design decision that enables the desired response. In this example, if the payment module uses the "Use an Intermediary" tactic (a payment gateway facade), then replacing the underlying gateway implementation requires changing only the facade's internal wiring – achieving the response measure of affecting at most 2 modules.

Without the tactic, the payment gateway might be referenced directly in 15 modules across the codebase, and replacing it would require modifying all 15 – violating the response measure and indicating a modifiability deficiency.

4.1.4 Quality Attribute Tactics Taxonomy

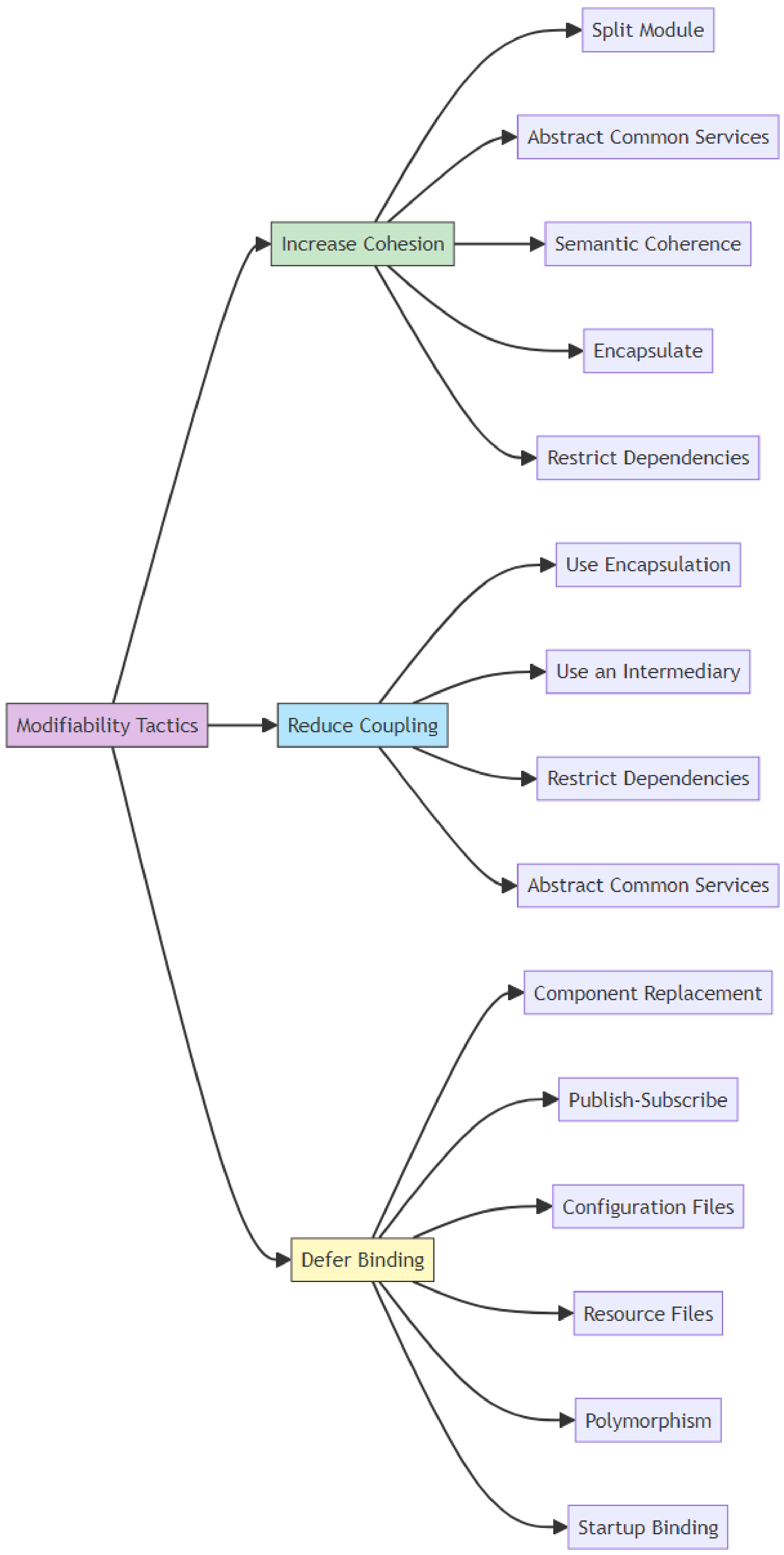
Bass et al. define tactics for seven quality attributes [1]:

Quality Attribute	Tactic Categories	Example Tactics
Availability	Detect Faults, Recover from Faults, Prevent Faults	Heartbeat, Voting, Active Redundancy, Checkpoint/Rollback
Interoperability	Locate Services, Manage Interfaces	Discover Service, Orchestrate, Tailor Interface
Modifiability	Increase Cohesion, Reduce Coupling, Defer Binding	Split Module, Use an Intermediary, Publish-Subscribe
Performance	Control Resource Demand, Manage Resources	Reduce Overhead, Manage Sampling Rate, Introduce Concurrency
Security	Resist Attacks, Detect Attacks, Recover from Attacks	Authenticate Users, Authorize Users, Encrypt Data
Testability	Observe/Control System State	Record/Playback, Abstract Data Sources, Sandbox
Usability	User Initiative, System Initiative	Cancel, Undo, Task Model, User Model

This thesis focuses exclusively on **modifiability tactics** because they (a) directly target ISO 25010 maintainability sub-characteristics, (b) are implementable through source code transformation, and (c) produce measurable changes in static analysis metrics.

4.2 The Modifiability Tactics Catalog

The following diagram shows the three categories of modifiability tactics and their key members:


Modifiability Tactics
Increase Cohesion
Split Module
Abstract Common Services
Semantic Coherence
Encapsulate
Restrict Dependencies
Reduce Coupling
Use Encapsulation
Use an Intermediary
Restrict Dependencies
Abstract Common Services
Defer Binding
Component Replacement
Publish-Subscribe
Configuration Files
Resource Files
Polymorphism
Startup Binding

Bass, Clements, and Kazman define 15 modifiability tactics organized into three categories based on the mechanism they use to improve modifiability [1]. The categories correspond to three fundamental strategies:

1. **Increase Cohesion** – ensure that things that change together are located together
2. **Reduce Coupling** – ensure that changes in one module do not propagate to others
3. **Defer Binding Time** – delay decisions so they can be changed without modifying code

For each tactic below, we provide: the definition, how it works mechanically, which ISO 25010 sub-characteristic it targets, and a concrete Python code example showing the transformation.

4.2.1 Category 1: Increase Cohesion (5 Tactics)

The cohesion tactics ensure that each module has a single, well-defined purpose. When cohesion is high, a change in requirements typically affects only one module because related responsibilities are co-located.

4.2.1.1 Tactic 1: Split Module

Definition: Break a module that has multiple responsibilities into smaller, focused modules, each with a single responsibility.

How it works: Identify distinct responsibilities within a module (e.g., data validation, business logic, persistence). Extract each responsibility into its own module with a clear interface. The original module may become a thin coordinator or may be eliminated entirely.

ISO 25010 target: Modularity, Modifiability

Example:

Before – a monolithic user service:

```python
# user_service.py -- handles everything user-related
class UserService:
    def validate_email(self, email):
        import re
        return bool(re.match(r'^[\w.-]+@[\w.-]+\.\w+$', email))

    def hash_password(self, password):
        import hashlib
        return hashlib.sha256(password.encode()).hexdigest()

    def save_user(self, user_data):
        # Direct database interaction
        conn = get_db_connection()
        conn.execute("INSERT INTO users ...", user_data)
        conn.commit()

    def send_welcome_email(self, email):
        import smtplib
        server = smtplib.SMTP('smtp.example.com')
        server.send_message(...)

    def register_user(self, name, email, password):
        if not self.validate_email(email):
            raise ValueError("Invalid email")
        hashed = self.hash_password(password)
        self.save_user({"name": name, "email": email, "password": hashed})
        self.send_welcome_email(email)
```

After – split into focused modules:

```
# validation.py
class UserValidator:
    def validate_email(self, email: str) -> bool:
        import re
        return bool(re.match(r'^[\w.-]+@[\w.-]+\.\w+$', email))

# security.py
class PasswordService:
    def hash_password(self, password: str) -> str:
        import hashlib
        return hashlib.sha256(password.encode()).hexdigest()

# user_repository.py
class UserRepository:
    def save(self, user_data: dict) -> None:
        conn = get_db_connection()
        conn.execute("INSERT INTO users ...", user_data)
        conn.commit()

# notification.py
class EmailNotificationService:
    def send_welcome_email(self, email: str) -> None:
        import smtplib
        server = smtplib.SMTP('smtp.example.com')
        server.send_message(...)

# user_service.py -- now a thin coordinator
class UserService:
    def __init__(self, validator, password_svc, repo, notifier):
        self.validator = validator
        self.password_svc = password_svc
        self.repo = repo
        self.notifier = notifier

    def register_user(self, name: str, email: str, password: str):
        if not self.validator.validate_email(email):
            raise ValueError("Invalid email")
        hashed = self.password_svc.hash_password(password)
        self.repo.save({"name": name, "email": email, "password": hashed})
        self.notifier.send_welcome_email(email)
```

The transformation reduces the cyclomatic complexity of each individual module, improves testability (each module can be tested independently with mock dependencies), and increases modularity (changing the email service does not require touching the password hashing logic).

4.2.1.2 Tactic 2: Abstract Common Services

Definition: Identify common functionality used across multiple modules and extract it into a shared service, base class, or utility module.

How it works: When two or more modules contain duplicated or near-duplicated logic, extract that logic into a common module and have both modules depend on it. This eliminates duplication and creates a single point of change.

ISO 25010 target: Modularity, Reusability

Example:

Before – duplicated logging and error handling in two services:

```
# order_service.py
class OrderService:
    def create_order(self, data):
        try:
            # business logic...
            import logging
            logging.getLogger("orders").info(f"Order created: {data['id']}")
            return {"status": "success", "data": result}
        except Exception as e:
            import logging
            logging.getLogger("orders").error(f"Order failed: {e}")
            return {"status": "error", "message": str(e)}

# inventory_service.py
class InventoryService:
    def update_stock(self, data):
        try:
            # business logic...
            import logging
            logging.getLogger("inventory").info(f"Stock updated: {data['sku']}")
            return {"status": "success", "data": result}
        except Exception as e:
            import logging
            logging.getLogger("inventory").error(f"Update failed: {e}")
            return {"status": "error", "message": str(e)}
```

After – common service extracted:

```
# common/service_base.py
import logging
from typing import Any, Callable

class ServiceBase:
    def __init__(self, logger_name: str):
        self.logger = logging.getLogger(logger_name)

    def execute(self, operation: Callable, success_msg: str,
                *args, **kwargs) -> dict:
        try:
            result = operation(*args, **kwargs)
            self.logger.info(success_msg)
            return {"status": "success", "data": result}
        except Exception as e:
            self.logger.error(f"Operation failed: {e}")
            return {"status": "error", "message": str(e)}

# order_service.py
class OrderService(ServiceBase):
    def __init__(self):
        super().__init__("orders")

    def create_order(self, data):
        return self.execute(
            self._do_create_order,
            f"Order created: {data['id']}",
            data
        )

    def _do_create_order(self, data):
        # pure business logic
        ...
```

This transformation eliminates code duplication (Visser's Guideline 3: Write Code Once), creates a single point of change for logging/error handling behavior, and improves reusability by making the common pattern available to all future services.

4.2.1.3 Tactic 3: Increase Semantic Coherence

Definition: Ensure that all responsibilities within a module are semantically related – that they serve a single, unified purpose.

How it works: Analyze the responsibilities of each module and identify responsibilities that are semantically unrelated to the module's primary purpose. Move those responsibilities to more appropriate modules. This differs from "Split Module" in that it focuses on *reassigning* misplaced responsibilities rather than decomposing a module entirely.

ISO 25010 target: Modularity, Analysability

Example:

Before – a `UserProfile` class that also manages authentication:

```python
# user_profile.py
class UserProfile:
    def get_display_name(self, user_id):
        ...  # Profile concern

    def update_avatar(self, user_id, image):
        ...  # Profile concern

    def verify_password(self, user_id, password):
        ...  # Authentication concern -- wrong module!

    def generate_api_token(self, user_id):
        ...  # Authentication concern -- wrong module!

    def get_profile_stats(self, user_id):
        ...  # Profile concern
```

After – coherent modules:

```python
# user_profile.py -- only profile responsibilities
class UserProfile:
    def get_display_name(self, user_id):
        ...
    def update_avatar(self, user_id, image):
        ...
    def get_profile_stats(self, user_id):
        ...

# authentication.py -- only auth responsibilities
class AuthenticationService:
    def verify_password(self, user_id, password):
        ...
    def generate_api_token(self, user_id):
        ...
```

4.2.1.4 Tactic 4: Encapsulate

Definition: Group related elements (data and the operations that manipulate that data) and expose only a well-defined interface to the rest of the system.

How it works: Identify data that is accessed directly by external modules. Wrap that data in a class or module with a controlled API. External modules interact only through the API, not with the underlying data structures or implementation.

ISO 25010 target: Modularity, Modifiability

Example:

Before – direct access to data structures:

```python
# Callers directly access and manipulate the internal dict
order = {"items": [], "total": 0, "status": "pending"}
order["items"].append({"sku": "ABC", "price": 29.99, "qty": 2})
order["total"] = sum(i["price"] * i["qty"] for i in order["items"])
order["status"] = "confirmed"
```

After – encapsulated with controlled interface:

```python
class Order:
    def __init__(self):
        self._items = []
        self._status = "pending"

    def add_item(self, sku: str, price: float, qty: int) -> None:
        self._items.append({"sku": sku, "price": price, "qty": qty})

    @property
    def total(self) -> float:
        return sum(i["price"] * i["qty"] for i in self._items)

    def confirm(self) -> None:
        if not self._items:
            raise ValueError("Cannot confirm empty order")
        self._status = "confirmed"

    @property
    def status(self) -> str:
        return self._status
```

Now the internal representation (a list of dicts) can be changed to a database-backed structure, a dataclass, or any other format without affecting any module that uses `Order`. The encapsulation boundary protects external modules from internal change.

4.2.1.5 Tactic 5: Restrict Dependencies

Definition: Limit which modules are permitted to depend on which other modules, enforcing architectural rules that prevent uncontrolled coupling.

How it works: Define explicit dependency rules (e.g., "presentation modules may not import data access modules directly; they must go through service modules"). Enforce these rules through conventions, linting rules, or architectural constraints. Remove or redirect any violating dependencies.

ISO 25010 target: Modularity, Testability

Example:

Before – uncontrolled imports creating a tangled dependency graph:

```python
# views/dashboard.py -- presentation layer importing directly from data layer
from database.queries import get_user_stats, get_order_history
from database.connection import get_raw_connection
from utils.formatters import format_currency
from services.cache import CacheManager
```

After – dependencies restricted to follow layering rules:

```python
# views/dashboard.py -- presentation layer imports only from service layer
from services.dashboard_service import DashboardService

class DashboardView:
    def __init__(self, dashboard_service: DashboardService):
        self.service = dashboard_service

    def render(self, user_id: str):
        stats = self.service.get_user_dashboard(user_id)
        return self.template.render(stats=stats)
```

```python
# services/dashboard_service.py -- service layer mediates access
from repositories.user_repository import UserRepository
from repositories.order_repository import OrderRepository

class DashboardService:
    def __init__(self, user_repo: UserRepository, order_repo: OrderRepository):
        self.user_repo = user_repo
        self.order_repo = order_repo

    def get_user_dashboard(self, user_id: str) -> dict:
        stats = self.user_repo.get_stats(user_id)
        orders = self.order_repo.get_recent(user_id)
        return {"stats": stats, "orders": orders}
```

4.2.2 Category 2: Reduce Coupling (4 Tactics)

Coupling tactics prevent changes from propagating across module boundaries. When coupling is low, modifying one module does not require modifying others – the change is contained.

4.2.2.1 Tactic 6: Use Encapsulation

Definition: Hide a module's internal implementation behind a stable public interface, so that changes to the implementation do not affect dependent modules.

How it works: Define a clear boundary between a module's interface (what it promises to do) and its implementation (how it does it). External modules depend only on the interface, never on implementation details. In Python, this often involves using abstract base classes, protocols, or well-defined public APIs.

ISO 25010 target: Modularity, Modifiability

Example:

Before – callers depend on a specific implementation:

```python
# Multiple modules directly use the Redis client
import redis

class OrderCache:
    def __init__(self):
        self.client = redis.Redis(host='localhost', port=6379)

    def get_order(self, order_id):
        data = self.client.get(f"order:{order_id}")
        return json.loads(data) if data else None
```

After – implementation hidden behind an interface:

```python
# cache/interface.py
from abc import ABC, abstractmethod

class CacheInterface(ABC):
    @abstractmethod
    def get(self, key: str) -> dict | None: ...

    @abstractmethod
    def set(self, key: str, value: dict, ttl: int = 3600) -> None: ...

# cache/redis_cache.py
class RedisCache(CacheInterface):
    def __init__(self, host='localhost', port=6379):
        self.client = redis.Redis(host=host, port=port)

    def get(self, key: str) -> dict | None:
        data = self.client.get(key)
        return json.loads(data) if data else None

    def set(self, key: str, value: dict, ttl: int = 3600) -> None:
        self.client.setex(key, ttl, json.dumps(value))

# cache/memory_cache.py -- alternative implementation
class MemoryCache(CacheInterface):
    def __init__(self):
        self._store = {}

    def get(self, key: str) -> dict | None:
        return self._store.get(key)

    def set(self, key: str, value: dict, ttl: int = 3600) -> None:
        self._store[key] = value
```

Now switching from Redis to an in-memory cache (or Memcached, or a database-backed cache) requires changing only the dependency injection configuration, not any module that uses the cache.

4.2.2.2 Tactic 7: Use an Intermediary

Definition: Introduce an indirection layer (broker, facade, mediator, proxy, adapter) between two modules so that they do not depend on each other directly.

How it works: When module A depends on module B, and this dependency creates problematic coupling (e.g., A would need to change whenever B changes), introduce module C between them. A depends on C, and C depends on B. Changes to B now only affect C, not A.

ISO 25010 target: Modularity, Modifiability

This is the tactic with the most service-oriented pattern implementations – Bogner found 9 SOA patterns realizing this tactic, including Service Facade, Proxy Capability, and Service Broker [18].

Example:

Before – direct dependency on external payment API:

```python
# order_processor.py
import stripe

class OrderProcessor:
    def charge_customer(self, amount, customer_id):
        return stripe.Charge.create(
            amount=int(amount * 100),
            currency="usd",
            customer=customer_id
        )
```

After – intermediary facade decouples from specific provider:

```python
# payment/gateway.py (intermediary)
from abc import ABC, abstractmethod

class PaymentGateway(ABC):
    @abstractmethod
    def charge(self, amount: float, currency: str,
               customer_ref: str) -> PaymentResult: ...

class StripeGateway(PaymentGateway):
    def charge(self, amount, currency, customer_ref):
        import stripe
        result = stripe.Charge.create(
            amount=int(amount * 100),
            currency=currency,
            customer=customer_ref
        )
        return PaymentResult(success=True, transaction_id=result.id)

class PayPalGateway(PaymentGateway):
    def charge(self, amount, currency, customer_ref):
        # PayPal-specific implementation
        ...

# order_processor.py -- depends on abstraction, not Stripe
class OrderProcessor:
    def __init__(self, gateway: PaymentGateway):
        self.gateway = gateway

    def charge_customer(self, amount, customer_id):
        return self.gateway.charge(amount, "usd", customer_id)
```

4.2.2.3 Tactic 8: Restrict Dependencies

(See Category 1, Tactic 5 above – this tactic appears in both the "Increase Cohesion" and "Reduce Coupling" categories in the literature because it serves both purposes: it increases cohesion by forcing modules to work within their layer, and it reduces coupling by eliminating unauthorized cross-layer dependencies.)

4.2.2.4 Tactic 9: Abstract Common Services

(See Category 1, Tactic 2 above – similarly, this tactic serves both cohesion and coupling goals. By extracting common functionality, it both increases the cohesion of individual modules, which no longer contain duplicated non-core logic, and reduces coupling by creating a stable shared dependency rather than ad-hoc inter-module coupling.)

4.2.3 Category 3: Defer Binding Time (6 Tactics)

Binding-time tactics postpone when design decisions take effect. The later a decision is bound, the more flexibility the system has to accommodate change without requiring code modification.

4.2.3.1 Tactic 10: Component Replacement

Definition: Design the system so that entire components can be replaced at deployment or runtime without modifying the rest of the system.

How it works: Define clear interfaces for components and use dependency injection, plugin architectures, or service registries to bind concrete implementations at runtime or deployment time.

ISO 25010 target: Modifiability, Reusability

4.2.3.2 Tactic 11: Publish-Subscribe

Definition: Decouple event producers from event consumers through an intermediary event bus or notification mechanism, so that producers do not need to know which consumers exist.

How it works: Instead of module A directly calling methods on modules B, C, and D when an event occurs, module A publishes an event to a bus. Modules B, C, and D subscribe to events they care about. Adding a new consumer (module E) requires no changes to the producer.

ISO 25010 target: Modularity, Modifiability

Example:

Before – tight coupling between order creation and side effects:

```
class OrderService:
    def __init__(self, inventory, email, analytics, loyalty):
        self.inventory = inventory
        self.email = email
        self.analytics = analytics
        self.loyalty = loyalty

    def create_order(self, order_data):
        order = self._save_order(order_data)
        self.inventory.reserve_items(order.items)      # direct call
        self.email.send_confirmation(order.customer)   # direct call
        self.analytics.track_purchase(order)           # direct call
        self.loyalty.award_points(order.customer, order.total)  # direct call
        return order
```

After – event-driven with publish-subscribe:

```
# events/bus.py
class EventBus:
    def __init__(self):
        self._subscribers = {}

    def subscribe(self, event_type: str, handler):
        self._subscribers.setdefault(event_type, []).append(handler)

    def publish(self, event_type: str, data):
        for handler in self._subscribers.get(event_type, []):
            handler(data)

# order_service.py -- publishes events, does not know about subscribers
class OrderService:
    def __init__(self, event_bus: EventBus):
        self.event_bus = event_bus

    def create_order(self, order_data):
        order = self._save_order(order_data)
        self.event_bus.publish("order.created", order)
        return order

# Each subscriber registers independently
event_bus = EventBus()
event_bus.subscribe("order.created", inventory_service.reserve_items)
event_bus.subscribe("order.created", email_service.send_confirmation)
event_bus.subscribe("order.created", analytics_service.track_purchase)
event_bus.subscribe("order.created", loyalty_service.award_points)
```

Adding a new side effect (e.g., fraud detection) requires only adding a new subscriber – zero changes to `OrderService`.

4.2.3.3 Tactic 12: Configuration Files

Definition: Extract hardcoded values and behavioral choices into external configuration files that are read at startup time.

How it works: Identify values embedded directly in source code (database URLs, feature flags, API keys, thresholds, algorithm selections) and move them to configuration files (YAML, JSON, environment variables, .ini files). The application reads these at startup and uses them to configure its behavior.

ISO 25010 target: Modifiability, Analysability

Example:

Before – hardcoded values scattered through code:

```
class DatabaseService:
    def connect(self):
        return psycopg2.connect(
            host="db.production.example.com",
            port=5432,
            database="myapp_prod",
            user="admin",
            password="s3cret_p@ss"
        )

class RateLimiter:
    MAX_REQUESTS = 100
    WINDOW_SECONDS = 60
```

After – externalized to configuration:

```
# config.yaml
database:
  host: ${DB_HOST}
  port: ${DB_PORT:-5432}
  name: ${DB_NAME}
  user: ${DB_USER}
  password: ${DB_PASSWORD}

rate_limiter:
  max_requests: 100
  window_seconds: 60
```

```
# config.py
import yaml
```

```python
import os

def load_config(path: str = "config.yaml") -> dict:
    with open(path) as f:
        config = yaml.safe_load(f)
    # Resolve environment variables
    return _resolve_env_vars(config)

# database_service.py
class DatabaseService:
    def __init__(self, config: dict):
        self.db_config = config["database"]

    def connect(self):
        return psycopg2.connect(**self.db_config)
```

4.2.3.4 Tactic 13: Resource Files

Definition: Externalize locale-specific, domain-specific, or frequently-changed content into resource files that can be modified without changing code.

ISO 25010 target: Modifiability

4.2.3.5 Tactic 14: Polymorphism

Definition: Use polymorphic dispatch (inheritance, protocols, or duck typing) so that the specific behavior invoked is determined by the type of the object at runtime, not by conditional logic in the caller.

ISO 25010 target: Modifiability

Example:

Before – conditional logic selects behavior:

```python
def calculate_shipping(order, method):
    if method == "standard":
        return order.weight * 0.5
    elif method == "express":
        return order.weight * 1.5 + 10
    elif method == "overnight":
        return order.weight * 3.0 + 25
    else:
        raise ValueError(f"Unknown method: {method}")
```

After – polymorphism delegates to the right implementation:

```python
from abc import ABC, abstractmethod

class ShippingStrategy(ABC):
    @abstractmethod
    def calculate(self, order) -> float: ...
```

```
class StandardShipping(ShippingStrategy):
    def calculate(self, order):
        return order.weight * 0.5

class ExpressShipping(ShippingStrategy):
    def calculate(self, order):
        return order.weight * 1.5 + 10

class OvernightShipping(ShippingStrategy):
    def calculate(self, order):
        return order.weight * 3.0 + 25

# Adding a new shipping method requires only a new class, no changes to callers
def calculate_shipping(order, strategy: ShippingStrategy):
    return strategy.calculate(order)
```

4.2.3.6 Tactic 15: Start-up Time Binding

Definition: Bind configuration and behavioral choices at application startup time rather than at compile time, allowing changes through configuration rather than code modification.

ISO 25010 target: Modifiability, Analysability

This tactic overlaps with "Configuration Files" but is broader – it includes any mechanism that defers binding to startup: reading environment variables, loading plugin modules dynamically, selecting database drivers based on configuration, etc.

4.2.4 Comprehensive Summary Table

The following table summarizes all 15 modifiability tactics with their measurability and LLM implementability assessment:

#	Tactic	Category	ISO 25010 Target	Measurable By	LLM Implementable?
1	Split Module	Increase Cohesion	Modularity, Modifiability	Per-module CC reduction, LOC per module, module count	**Yes** – Extract Class/Method is a well-understood refactoring
2	Abstract Com-mon Services	Increase Cohesion	Modularity, Reusability	Duplication %, total LOC reduction, shared module count	**Yes** – duplication detection and extraction is within LLM capability

#	Tactic	Category	ISO 25010 Target	Measurable By	LLM Implementable?
3	Increase Semantic Coherence	Increase Cohesion	Modularity, Analysability	LCOM (Lack of Cohesion), responsibility distribution	**Yes** – LLMs can analyze and reassign responsibilities
4	Encapsulate	Increase Cohesion	Modularity, Modifiability	Public API surface, hidden code %	**Yes** – introducing classes/interfaces from raw data
5	Restrict Dependencies	Increase Cohesion / Reduce Coupling	Modularity, Testability	Circular dependency count, import graph depth	**Yes** – LLMs can reorganize imports and introduce interfaces
6	Use Encapsulation	Reduce Coupling	Modularity, Modifiability	CBO, afferent/efferent coupling	**Yes** – introducing abstract interfaces for concrete deps
7	Use an Intermediary	Reduce Coupling	Modularity, Modifiability	CBO, fan-in/fan-out, intermediary count	**Yes** – facade/adapter introduction is a standard refactoring
8	Maintain Existing Interface	Reduce Coupling	Modifiability, Analysability	API breaking change count	**Partial** – relevant mainly during API evolution, not snapshot refactoring
9	Anticipate Expected Changes	Increase Cohesion	Modifiability	Change propagation analysis	**No** – requires domain knowledge about future changes
10	Generalize Module	Increase Cohesion	Reusability, Modifiability	Parameter genericity, reuse frequency	**Partial** – risk of over-engineering without usage context
11	Publish-Subscribe	Defer Binding	Modularity, Modifiability	Direct dependency count, event channel count	**Yes** – event bus refactoring is well-defined
12	Configuration Files	Defer Binding	Modifiability, Analysability	Hardcoded value count, config externalization ratio	**Yes** – extracting hardcoded values is straightforward
13	Resource Files	Defer Binding	Modifiability	Resource externalization ratio	**Yes** – simple extraction task

#	Tactic	Category	ISO 25010 Target	Measurable By	LLM Implementable?
14	Polymorphism	Defer Binding	Modifiability	Conditional complexity reduction, strategy pattern count	**Yes** – Replace Conditional with Polymorphism is a classic refactoring
15	Start-up Time Binding	Defer Binding	Modifiability, Analysability	Hardcoded config count, env-var usage	**Yes** – similar to configuration file extraction

The "LLM Implementable?" column is critical for the thesis. Of the 15 tactics, **11 are assessed as fully implementable** by an LLM through source code transformation, 2 are partially implementable, and 2 are not implementable without external domain knowledge. The thesis selects 8 tactics for its pipeline based on three criteria: (1) full LLM implementability, (2) measurability through static analysis, and (3) demonstrated structural compatibility with common backend patterns.

4.3 Pattern-Tactic Interactions

Tactics do not exist in isolation. In real systems, they are implemented within the context of existing architectural patterns. Understanding how tactics interact with patterns is essential for predicting whether a tactic implementation will succeed or fail.

4.3.1 Harrison & Avgeriou's Interaction Model

Harrison and Avgeriou developed the definitive model for pattern-tactic interactions, based on analysis of reliability tactics across multiple patterns and validated through three industrial case studies [11].

4.3.1.1 Five Structural Interaction Types

When a tactic is implemented within a pattern, it causes structural changes to the pattern's components and connectors. Harrison identifies five types, ordered by increasing impact:

Type	Description	Component Change	Connector Change	Impact Level	Example
Implemented-in	Tactic behavior is added inside an existing component; no structural change to the pattern	Modified internals	None	Lowest	Adding input validation logic inside an existing Controller in MVC
Replicates	An existing component is duplicated to realize the tactic	Duplicated component	New connectors to replica	Low-Medium	Creating a read-only database replica for load distribution
Add-in-pattern	A new component is added within the existing pattern structure	New component within pattern	New connectors within pattern	Medium	Adding a Service Facade between existing layers in a Layered architecture
Add-out-of-pattern	A new component is added outside the pattern structure, crossing pattern boundaries	New component outside pattern	Connectors cross pattern boundary	High	Adding a monitoring service that bypasses the normal layered structure

Type	Description	Component Change	Connector Change	Impact Level	Example
Modify	An existing component is fundamentally altered in structure	Fundamentally changed	May be altered	Highest	Splitting a monolithic data store into separate per-service databases

4.3.1.2 Impact Magnitude Scale

Harrison also provides a five-point qualitative scale for assessing how well a tactic fits within a given pattern:

Rating	Meaning	Criteria
Good Fit (++)	Tactic integrates naturally into the pattern	1-2 component changes, no new connectors outside pattern
Minor Changes (+)	Tactic requires small modifications to the pattern	3 component changes, minor connector additions
Neutral (~)	Neither naturally fitting nor conflicting	Mixed impact; some beneficial, some neutral
Significant Changes (-)	Tactic requires substantial pattern modifications	Multiple component changes and new connectors
Poor Fit (–)	Tactic conflicts with the pattern's structure	Requires adding components outside pattern, causing drift

The practical threshold is important: **3 or fewer participant (component) changes indicate lower impact; more than 3 indicate higher impact** [11].

4.3.1.3 Behavioral Interaction Types

Beyond structural changes, tactics also affect the behavioral sequences within patterns:

1. **Adding action sequences** – New sequences of actions are introduced within or outside existing behavioral flows
2. **Timing modifications** – Four sub-types:
    - Adding new explicit timing constraints
    - Adding new implicit timing constraints
    - Changing existing explicit timing
    - Changing existing implicit timing

4.3.1.4 Architecture Drift as Snowball Effect

Harrison's most important practical finding is that tactic implementation can trigger cascading structural changes – what he calls a "snowball effect." In one case study, adding a layer bypass for performance improvement required:

1. A new component outside the normal layer structure (Add-out-of-pattern)
2. A duplicate authorization component because the bypassed layer contained authorization logic
3. Duplicated code between the original and duplicate authorization components

The result: a tactic intended to improve performance *degraded* maintainability through code duplication and structural irregularity. Harrison's warning is direct:

> "Implementing tactics is actually a form of architecture drift." [11]

This finding has critical implications for LLM-based tactic implementation. The pipeline must be designed to prefer tactic implementations that are "Implemented-in" or "Add-in-pattern" types (lower impact) and to flag or avoid implementations that would create "Add-out-of-pattern" changes (high drift risk).

4.3.2 Pattern-Tactic Compatibility Examples

The following table illustrates how the same tactic can have very different compatibility ratings depending on the target pattern:

Tactic	MVC	Layered	Pipe-and-Filter	Repository	Broker
Split Module	Good Fit (++) – natural decomposition within each layer	Good Fit (++) – splits within a layer	Good Fit (++) – each filter is naturally isolated	Neutral (~)	Minor (+)

Tactic	MVC	Layered	Pipe-and-Filter	Repository	Broker
Use an Intermediary	Minor (+) – add a service layer between controller and model	Good Fit (++) – intermediary layers are native to the pattern	Poor Fit (–) – intermediaries break the pipeline flow	Good Fit (++) – repository itself is an intermediary	Good Fit (++) – broker is an intermediary
Publish-Subscribe	Minor (+) – replace controller-model callbacks with events	Significant (-) – events cross layer boundaries	Good Fit (++) – filters already communicate through channels	Minor (+)	Good Fit (++) – broker supports pub-sub natively

4.3.3 Kassab's Empirical Finding

The theoretical pattern-tactic interaction model is grounded in empirical reality by Kassab et al.'s survey of 809 software professionals [19]:

- **63% of real projects modify architectural patterns with tactics** during implementation
- Functionality (not quality requirements) is the primary driver for pattern selection
- Patterns are almost never applied "by the book" – they are adapted to the project's needs

This confirms that the pattern-tactic interaction problem is not academic. Every real system that uses architectural patterns has already applied tactics (whether the developers called them that or not), and understanding these interactions is essential for any automated approach that modifies system architecture.

4.4 Tactics in Service-Oriented and Microservice Systems

Bogner, Wagner, and Zimmermann provide the most detailed mapping of modifiability tactics to modern service-oriented architectures [18]. Their work systematically maps all 15 modifiability tactics onto principles and design patterns of both SOA and Microservice-Based Systems.

4.4.1 Mapping Methodology

Bogner's team compiled: - 15 modifiability tactics from Bass et al. (2003, 2012) and Bachmann et al. (2007) - 8 SOA principles from Erl - 8 Microservices principles from Lewis and Fowler - 118 SOA patterns from Erl and Rotem-Gal-Oz - 42 Microservices patterns from Richardson's catalog

They then performed a qualitative mapping, identifying which service-oriented patterns realize which modifiability tactics. The complete mapping data is publicly available on GitHub (xjreb/research-modifiability-tactics).

4.4.2 Key Results

Metric	SOA	Microservices
Total patterns in catalog	118	42
Patterns mapped to modifiability tactics	47 (~40%)	21 (50%)
Principle-to-tactic mappings	26 of 120 possible	15 of 120 possible
Principle with most tactic mappings	Service Loose Coupling (7 tactics)	Evolutionary Design (5 tactics)
Dominant tactic category (principles)	Reduce Coupling (16 mappings)	Reduce Coupling (7 mappings)
Dominant tactic category (patterns)	Reduce Coupling (23 patterns, ~49%)	Defer Binding Time (11 patterns, ~52%)
Tactic with most pattern matches	Use an Intermediary (9 SOA patterns)	Runtime Registration & Dynamic Lookup (5 MS patterns)
Tactic with zero pattern matches	Compile Time Binding (neither SOA nor MS)	Compile Time Binding (neither SOA nor MS)

4.4.3 Tactic Category Distribution

Tactic Category	SOA Pattern %	Microservices Pattern %	Strategy
Increase Cohesion	24% (11 patterns)	14% (3 patterns)	Underrepresented in both
Reduce Coupling	49% (23 patterns)	33% (7 patterns)	Dominant in SOA

Tactic Category	SOA Pattern %	Microservices Pattern %	Strategy
Defer Binding Time	28% (13 patterns)	52% (11 patterns)	Dominant in Microservices

4.4.4 Strategic Differences

The data reveals a fundamental strategic difference in how the two architectural styles achieve modifiability:

- **SOA** achieves modifiability through **governance, standardization, and intermediaries**. The heavy use of Reduce Coupling tactics (49% of patterns) reflects SOA's emphasis on well-defined service contracts, standardized protocols (SOAP, WSDL), and intermediary patterns (Service Facade, Service Broker, Enterprise Service Bus) that decouple services through centralized mediation.
- **Microservices** achieve modifiability through **evolutionary design, infrastructure automation, and runtime discovery**. The dominance of Defer Binding Time tactics (52% of patterns) reflects Microservices' emphasis on independent deployability, dynamic service discovery, event-driven communication, and infrastructure-as-code.

4.4.5 The Cohesion Gap

A particularly striking finding is that **only 3 of 21 mapped Microservices patterns fall under "Increase Cohesion"** – despite small, cohesive services being a core Microservices philosophy. Bogner notes that this gap may indicate that cohesion in Microservices is achieved through the decomposition philosophy itself (each service is inherently cohesive because it owns a single bounded context) rather than through specific patterns.

This cohesion gap represents an area where LLM-driven tactic implementation could add particular value: helping developers decompose services that have grown beyond their original bounded context back into cohesive units, using Split Module and Increase Semantic Coherence tactics.

4.5 The Tactic Research Landscape

Marquez, Astudillo, and Kazman provide the most comprehensive overview of the state of architectural tactics research through their systematic mapping study of 91 primary studies spanning 2003-2021 [20].

4.5.1 Scale and Scope

Dimension	Value
Studies screened	552 candidates from 7 digital libraries

Dimension	Value
Primary studies selected	91 (79 database search + 12 snowballing)
Time span	2003-2021 (18 years)
Digital libraries	IEEE Xplore, SpringerLink, Scopus, ACM, Web of Science, ScienceDirect, Wiley
Quality attributes covered	12 (original 7 + adaptability, dependability, reliability, deployability, scalability, fault tolerance, safety)

4.5.2 The Rigor Gap

The most sobering finding is the pervasive lack of methodological rigor:

Methodological Aspect	Studies Lacking It	Percentage
Tactic identification method not described	65 of 91	**71%**
Data source for recognizing tactics not described	63 of 91	**69%**
Tactic description/characterization mechanism not described	54 of 91	**59%**

Marquez concludes:

> "Little rigor has been used to characterize and define architectural tactics; most architectural tactics proposed in the literature do not conform to the original definition." [20]

4.5.3 Quality Attribute Coverage Imbalance

Quality Attribute	# of Dedicated Studies	Research Maturity
Security	18	Well-studied
Fault Tolerance	5	Moderate
Availability	4	Moderate
Performance	4	Moderate
Safety	4	Moderate
Modifiability	**1**	**Severely under-researched**
Testability	0	Not studied

The fact that modifiability – the quality attribute most directly related to ISO 25010 maintainability – has received exactly one dedicated study out of 91 is a critical research gap. This thesis directly addresses this gap.

4.5.4 Detection Tools and Techniques

The mapping study catalogues existing tactic detection approaches:

Technique	# of Studies	Tools
Manual mapping	10	Expert judgment
Code analysis	10	Custom scripts, AST analysis
Text analysis / NLP	4	BERT classifiers, LDA topic modeling
Multifacetic (combined)	3	Archie, ArchEngine, ARCODE
Machine learning	Subset of above	SVM, Decision Trees, Bayesian LR, AdaBoost
Not described	65	N/A

Critically, **all existing tools detect tactics but none implement them**. The detection-to-implementation gap is the central motivation for the thesis. As Marquez notes:

> "Tactics have become relevant to map architecture decisions onto source code, and automation opportunities beckon within reach." [20]

4.5.5 Research Type Distribution

Research Type	Percentage
Solution proposals (new methods/tools)	47.3%
Evaluation research (applying methods)	39.6%
Philosophical / experience papers	13.1%

Contribution Type	Percentage
Frameworks	48.4%
Models	16.5%
Guidelines	15.4%
Lessons learned	7.7%
Theories	6.6%

The dominance of frameworks and solution proposals indicates a field still in the theory-building phase. The thesis contributes to moving the field toward implementation and empirical validation.

4.6 Quality-Driven Architecture Development

4.6.1 Kim's Feature Model Approach

Kim, Kim, Lu, and Park proposed the most rigorous formalization of architectural tactics, using feature models to capture tactic variability and the Role-Based Metamodeling Language (RBML) to specify structural and behavioral semantics [17].

4.6.1.1 Feature Models for Tactic Relationships

Feature models capture five types of relationships between tactics:

Relationship	Meaning	Example
Mandatory	If the parent tactic category is selected, this tactic must be included	Fault Detection is mandatory for any Availability strategy
Optional	May or may not be included	State Resynchronization is optional within Recovery
Requires	Selecting tactic A requires also selecting tactic B	Active Redundancy requires Heartbeat for failure detection
Suggested	Selecting tactic A makes tactic B advisable but not required	Checkpoint/Rollback suggests State Resynchronization for faster recovery
Mutually exclusive	Selecting tactic A prevents selecting tactic B	Active Redundancy and Passive Redundancy are mutually exclusive

4.6.1.2 RBML Specifications

For each formalized tactic, Kim provides: - **Structural specification:** UML class diagram roles showing required components, interfaces, and relationships - **Behavioral specification:** Sequence diagram showing the expected interaction pattern - **Realization multiplicity:** How many instances of each role are valid (e.g., 1 FIFO queue vs. 3 FIFO queues)

4.6.1.3 Coverage and Limitations

Kim successfully formalized tactics for three quality attributes:

Quality Attribute	Tactics Formalized
Availability	Fault Detection (Ping/Echo, Heartbeat, Exception), Recovery Reintroduction (Checkpoint/Rollback, State Resynchronization), Recovery Preparation and Repair (Voting, Active Redundancy, Passive Redundancy)
Performance	Resource Arbitration (FIFO, Fixed Priority Scheduling, Dynamic Priority Scheduling), Resource Management (Introduce Concurrency, Maintain Multiple Copies/Cache)
Security	Resisting Attacks (Authentication: ID/Password, One-time Password; Authorization; Maintain Data Confidentiality), Recovering from Attacks (Restoration)

Modifiability is explicitly listed as future work. Kim notes:

> "It should be noted that not all tactics can be specified in the RBML. For instance, the resource demand tactics, which are concerned with managing resource demand, are difficult to formalize in the RBML due to the abstract nature of their solutions." [17]

This is a key gap the thesis addresses. While formal RBML specifications for modifiability tactics do not exist, the feature model relationship types (requires, suggested, mutually exclusive) are transferable and can be encoded as structured constraints in the LLM prompt chain. For example: - "Split Module" **requires** that the target module has multiple identifiable responsibilities - "Use an Intermediary" **suggests** prior application of "Use Encapsulation" - "Publish-Subscribe" and direct method invocation are **mutually exclusive** for the same interaction

4.6.2 Rahmati's Pattern Evaluation Framework

Rahmati and Tanhaei provide a complementary approach at the pattern level [21]. Instead of formalizing individual tactics, they evaluate how well entire architectural patterns support maintainability.

4.6.2.1 The Maintainability Hexagon

Their quality model decomposes maintainability into six sub-attributes, each scored on a 0-4 scale:

Score	Meaning
0	Large reduction in this sub-attribute
1	Relative reduction
2	Neutral

Score	Meaning
3	Relative increase
4	Large increase

4.6.2.2 Pattern Radar Scores

Pattern	Testability	Changeability	Understandability	Portability	Stability	Analysability
Multi-Layered	4	4	2	4	3	3
MVC	4	4	3	4	2	3
Broker	4	4	2	3	2	4
Object-Oriented	3	3	3	3	3	3
Repository	3	3	3	2	4	3
Pipe & Filter	1	4	0	3	2	0
Blackboard	1	4	0	1	2	0

Multi-Layered and Broker patterns achieve the highest overall maintainability scores. Pipe & Filter excels in changeability but scores poorly in testability, understandability, and analysability – a cautionary example that optimizing for one sub-attribute can come at the cost of others.

4.6.2.3 Case Study Evidence

Rahmati's two case studies provide critical empirical evidence:

Case Study 1 (Successful): A university research management system was refactored from a Proxy-based architecture to MVC. The result: - **50-64% average reduction in maintenance effort** across multiple maintenance scenarios - Improvement increased over time (12-14% additional improvement in subsequent months) as the team became more familiar with the new structure

Case Study 2 (Failed): A commercial PHP CMS was refactored from flat, unstructured code to MVC using the Yii Framework. The result: - **~10% increase in maintenance effort** – the refactoring made things worse - The system was already simple enough that the MVC overhead added complexity without proportional benefit - The team was unfamiliar with the target framework, adding a learning curve

The lesson is sharp:

> "Architecture patterns are preventing rather than preserving in maintainability aspects." [21]

This means that patterns *prevent* maintainability problems from arising in appropriate contexts, but they cannot *guarantee* maintainability if the lower-level implementation is flawed or if the pattern is misapplied. For the thesis, this finding reinforces that LLM-driven tactic implementation must:

1. **Assess the current system's complexity** before recommending architectural changes
2. **Match tactics to the target architecture** – not all tactics improve all systems
3. **Ensure correct lower-level realization** – the LLM must generate not just structurally correct but semantically correct code

4.6.3 Connecting Kim and Rahmati

The relationship between Kim's tactic-level formalization and Rahmati's pattern-level evaluation provides a layered decision framework:

1. **Rahmati's radar model** identifies *which maintainability sub-attributes are deficient* in the current system
2. **Bass's tactic catalog** identifies *which tactics target those specific sub-attributes*
3. **Kim's feature model relationships** constrain *which tactic combinations are valid*
4. **Harrison's interaction model** predicts *how well those tactics fit within the current architecture pattern*

This layered approach is what the thesis LLM pipeline must encode in its prompt chain: detect the architecture, assess the quality deficiencies, select appropriate tactics, verify pattern compatibility, and then implement.

4.7 Summary

This chapter established the theoretical foundation for architectural tactics and their role in improving software maintainability. The key takeaways are:

1. **Architectural tactics are fine-grained building blocks** that target single quality attribute responses, distinct from patterns (which package multiple tactics with built-in tradeoffs) and design techniques (which implement tactics at the code level) [1].

2. **The modifiability catalog contains 15 tactics** in three categories: Increase Cohesion (5), Reduce Coupling (4), and Defer Binding Time (6). Of these, 11 are assessed as fully implementable by an LLM through source code transformation.

3. **Pattern-tactic interactions vary widely** from Good Fit to Poor Fit, and poorly chosen tactic implementations can cause architecture drift with cascading maintainability degradation [11]. LLM-based implementation must respect pattern boundaries.

4. **SOA and Microservices achieve modifiability through different strategies**: SOA through governance and intermediaries (Reduce Coupling dominant), Microservices through evolutionary design and runtime discovery (Defer Binding Time dominant) [18].

5. **The research landscape has critical gaps**: modifiability has received only 1 dedicated study out of 91, 71% of studies lack methodological rigor in tactic identification, and no existing tool implements tactics automatically [20].

6. **Kim's feature model formalization** provides composability rules for tactics, but does not cover modifiability – a gap this thesis addresses [17].

7. **Rahmati's case studies** demonstrate that architecture-level refactoring can improve maintainability by 50-64% when well-matched, but can degrade it by 10% when misapplied [21] – underscoring the need for systematic, context-aware tactic selection.

8. **The detection-implementation gap** is the central problem: existing tools can detect tactics in code but cannot implement them. The thesis proposes that LLMs, augmented with architectural context and structured tactic specifications, can bridge this gap for maintainability tactics.

Chapter 5

Architecture Erosion & Drift

Learning objectives. After reading this chapter you will be able to (1) define architecture erosion and architecture drift and explain the difference between them, (2) describe the causes, symptoms, and consequences of erosion using Li et al.'s taxonomy, (3) explain why detection alone does not solve erosion using empirical evidence from the Rosik et al. case study, (4) summarize how practitioners discuss architectural tactics informally based on Bi et al.'s Stack Overflow mining study, and (5) articulate the detection-remediation disconnect that motivates automated architectural improvement.

5.1 What Is Architecture Erosion?

Every software system begins with an *intended architecture* — a set of design decisions about how modules, layers, and components should interact. Over time, the *implemented architecture* (what is actually in the code) diverges from that intent. This divergence is the broad phenomenon of **architecture erosion**.

5.1.1 A Four-Perspective Definition

Li et al. [4] conducted the first systematic mapping study dedicated exclusively to architecture erosion (AEr), analyzing 73 primary studies published between 2006 and 2019. One of their most important contributions is a refined definition that unifies four complementary perspectives found across the literature:

Perspective	Definition	Studies
Violation	The implemented architecture violates rules or constraints of the intended architecture	30
Structure	Internal structural flaws accumulate (e.g., cyclic dependencies, god classes)	9

Perspective	Definition	Studies
Quality	Observable quality attributes (maintainability, performance) degrade over time	6
Evolution	The architecture becomes resistant to change; modifications are increasingly costly	3

The synthesized definition reads:

> "Architecture erosion happens when the implemented architecture violates the intended architecture with flawed internal structure or when architecture becomes resistant to change." [4]

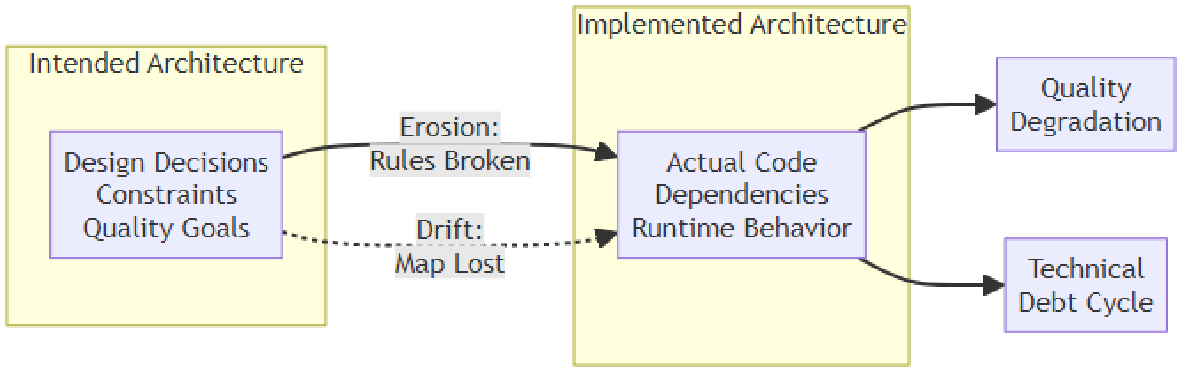
This multi-perspective view is important because it means erosion is not just about breaking rules. A system can be "correct" with respect to its layer boundaries and still suffer erosion if its quality metrics are degrading or if every change requires touching dozens of files. Students should internalize this: **erosion is a spectrum, not a binary**.

5.1.2 Erosion vs. Drift

Although the terms are sometimes used interchangeably, they capture different failure modes:

- **Architecture Erosion** is the violation of the intended architecture. Explicit rules, constraints, or design decisions are broken. For example, a layered architecture specifies that controllers must not access the database directly — if a developer adds a direct SQL call in a controller, that is erosion.
- **Architecture Drift** is the gradual, often imperceptible divergence where the implemented architecture no longer matches the design intent, even though no single rule may be obviously broken. The system "drifts" because small, individually reasonable decisions accumulate into an architecture that nobody designed.

Think of erosion as *breaking the rules* and drift as *losing the map.* Both lead to the same outcome — an architecture that is harder to maintain, extend, and reason about — but they require different detection strategies. Erosion can be caught by conformance checkers that compare code against explicit rules. Drift requires comparing the actual dependency structure against a high-level model of what the architecture *should* look like.



5.1.3 Causes of Erosion

Li et al. classify 13 categories of erosion causes into three groups [4]:

Technical causes:

Cause	Frequency	Description
Architecture violation	24.7%	Direct violations of intended architecture constraints
Evolution issues	23.3%	System grows in ways the original architecture did not anticipate
Technical debt	17.8%	Shortcuts and workarounds that accumulate over time

Non-technical causes:

Cause	Frequency	Description
Knowledge vaporization	15.1%	Original design rationale is lost as team members leave or documentation becomes stale
Organizational factors	Moderate	Communication gaps between teams, silos, unclear ownership
Time pressure	Moderate	Deadlines force shortcuts that violate architecture constraints

Mixed causes:

Technology evolution (new frameworks, language features, platform changes) and requirements churn create situations where both technical and organizational forces push the system away from its intended architecture.

Knowledge vaporization deserves special attention. It refers to the phenomenon where the *reasons* behind architectural decisions are lost. The code remains, but nobody remembers *why* a particular module boundary exists, *why* a certain dependency was forbidden, or *what* quality attribute a particular structure was protecting. When this knowledge evaporates, developers unknowingly make changes that violate the intent.

5.1.4 Consequences

The consequences of erosion are severe and empirically documented:

- **83.8% of consequence-mentioning studies report quality degradation** [4]. The most affected quality attributes are maintainability, evolvability, and extensibility.
- Erosion increases defect rates, reduces developer productivity, and makes the system brittle.
- Li et al. identify a **technical debt vicious cycle**: technical debt causes erosion, and erosion generates more technical debt. Once this cycle begins, it accelerates.

  "Technical debt is both a cause and consequence of AEr, forming a vicious cycle." [4]

5.1.5 The Tool Landscape

Li et al. catalog **35 tools** for erosion detection across the 73 studies. However, the landscape has significant limitations:

- **57.1% of tools** are based on Architecture Conformance Checking (ACC) — they compare actual dependencies against allowed ones.
- **Over 50% support only one programming language**, and that language is overwhelmingly Java.
- There is a major **research-practice gap**: 82.2% of erosion studies come from academia, and developers report that they do not use dedicated erosion-detection tools in practice.

5.1.6 Example: Layer Violations in a Web Application

Consider a standard 3-layer backend architecture:

```
Intended architecture:

    +-----------------+
    |  Controller     |  <- Handles HTTP requests
    +--------+--------+
             | (allowed)
             v
    +-----------------+
    |   Service       |  <- Contains business logic
    +--------+--------+
             | (allowed)
             v
    +-----------------+
    |  Repository     |  <- Handles database access
    +-----------------+
```

The intended architecture enforces a strict rule: **each layer may only depend on the layer directly below it**. The Controller layer calls Service methods; the Service layer calls Repository methods. The Controller should never access the Repository directly.

Now consider what happens under time pressure. A developer needs to display a simple count on a dashboard. Writing a service method feels like overhead for a one-line query, so they write:

```python
# controller/dashboard.py  -- EROSION: direct repository access
from repository.user_repo import UserRepository

class DashboardController:
    def get_user_count(self, request):
        repo = UserRepository()
        count = repo.count_all()  # Bypasses service layer!
        return {"user_count": count}
```

This is a **layer violation** — a textbook case of architecture erosion. The Controller now depends directly on the Repository, bypassing the Service layer. The correct implementation routes through a service:

```python
# controller/dashboard.py  -- CORRECT: goes through service layer
from service.user_service import UserService

class DashboardController:
    def get_user_count(self, request):
        service = UserService()
        count = service.get_user_count()
        return {"user_count": count}

# service/user_service.py
from repository.user_repo import UserRepository

class UserService:
    def get_user_count(self):
        repo = UserRepository()
        return repo.count_all()
```

One violation seems harmless. But when dozens of controllers bypass services, the service layer loses its purpose, business logic scatters across controllers, and the architecture erodes to the point where refactoring becomes prohibitively expensive.

5.2 Architecture Drift in Practice

5.2.1 The IBM Dublin Case Study

While Li et al. provide the theoretical taxonomy, Rosik et al. [8] provide the empirical ground truth. They conducted a **2-year longitudinal case study** at IBM Dublin Software Lab, studying the development of DAP 2.0 (Domino Application Portlet), a commercial system of approximately **28,500 lines of code** (16 packages, 95 classes).

What makes this study exceptional is its design:

- It is **longitudinal** (November 2005 – October 2007), not a one-time snapshot.
- It is conducted **in vivo** — on a real commercial project during active development, not on a student project or a post-hoc analysis of an open-source codebase.
- It uses **Reflexion Modelling**, a technique that compares the *planned* architecture (a high-level model drawn by the architect) against the *actual* architecture (dependencies extracted from the source code). Discrepancies appear as *divergent edges* — connections that exist in the code but should not exist according to the plan.

5.2.2 Key Findings

Over 6 evaluation sessions spanning 2 years, the researchers found:

Metric	Value
Total architectural inconsistencies	9 divergent edges reflecting 15 violating source-code relationships
Violations found in first session	5 (drift begins immediately, even during initial development)
Violations fixed by developers	**0 out of 9**
Convergent edges hiding violations	5 of 8 analyzed convergent edges contained hidden divergent relationships
Worst hidden case	1 convergent edge masked 41 out of 44 inconsistent relationships

The most striking finding is that **none of the 9 identified violations were ever fixed**. The researchers detected them, presented them to the developers, and the developers chose — deliberately — not to fix them.

5.2.3 Why Violations Persist: Developer Voices

The think-aloud sessions and retrospective interviews captured revealing developer statements:

> "You've seen my surprise the last time, when we ran the tool against the existing code base." — Developer, expressing genuine surprise at the violations [8]

Surprise indicates that the violations were **unintentional** — the developers did not know they were breaking the architecture. Yet even after learning about the violations, they did not fix them. The reasons are illuminating:

> "Maybe trying to fix these 'minor' issues would've possibly caused larger issues to appear and so made them not worth exploring…" [8]

> "If performance can be gained by breaking an architectural design, then this can sometimes be acceptable… time pressure is probably the main factor in some of the decision making that goes on during such a development." [8]

These quotes reveal three distinct barriers to remediation:

1. **Risk aversion**: Fixing a violation might introduce bugs elsewhere (ripple effects).

2. **Cost-benefit calculus**: Minor violations are perceived as low-risk and not worth the effort.
3. **Time pressure**: Fixing architectural issues is never "in scope" when there are features to deliver.

5.2.4 The Batch Detection Problem

The study used **batch processing** — evaluation sessions were conducted at roughly 4–5 month intervals. This schedule proved inadequate. By the time violations were detected, they had been present for months, other code had been built on top of them, and removing them would have required significant rework.

Rosik et al. conclude that **continuous, lightweight monitoring** is far more effective than periodic reviews. If violations are caught within hours or days, they can be fixed while the developer still has context and before dependent code accumulates. This finding aligns with modern CI/CD practices where static analysis runs on every commit.

5.2.5 False Negatives in Reflexion Modelling

A particularly troubling finding is that Reflexion Modelling itself can **conceal violations**. When a planned connection exists between two modules (a *convergent edge*), any additional unplanned connections between the same modules are hidden within the aggregate. Rosik et al. found that 5 out of 8 convergent edges contained hidden divergent relationships. In the worst case, a single convergent edge masked 41 out of 44 inconsistent relationships.

This means the 9 detected violations are likely an **undercount**. The real number of architectural inconsistencies was almost certainly higher, but the detection technique itself obscured them.

5.3 Practitioner Knowledge About Tactics

5.3.1 Mining Stack Overflow

If erosion is pervasive and detection tools are underused, how do practitioners actually learn about and discuss architectural tactics? Bi et al. [22] investigated this question by mining Stack Overflow, the largest Q&A platform for developers.

Their approach was semi-automatic:

1. **Dictionary training**: Used Word2vec on 2,301 Stack Overflow posts tagged with "software architecture" or "software design" to build a semantic dictionary of architecture tactic terms and their synonyms.
2. **Classification**: Trained six machine learning classifiers on 1,165 manually labeled posts. The best performer was **SVM with the trained dictionary**, achieving an F-measure of **0.865**.
3. **Mining**: Applied the classifier to the full Stack Overflow corpus (2012–2019), extracting 5,103 candidate posts, of which **4,195 were manually verified** as genuine QA-tactic discussions (82.2% precision).

4. **Analysis**: Mapped the mined posts to 21 architecture tactics and 8 quality attributes (following ISO 25010).

5.3.2 What Practitioners Discuss

The most-discussed quality attributes and tactics on Stack Overflow reveal clear practitioner priorities:

Quality Attribute	Instances	Share
Performance	1,725	41.1%
Security	987	23.5%
Reliability	612	14.6%
Maintainability	398	9.5%
Other (Compatibility, Usability, Portability, Functional Suitability)	473	11.3%

Most-Discussed Tactics	Instances
Time out	470
Authentication	389
Resource pooling	356
Heartbeat	298
Checkpoint/Rollback	245

Performance and Security tactics dominate Stack Overflow discussions. Maintainability tactics receive less attention, which is consistent with the general observation that maintainability is often a *non-functional* concern that developers defer in favor of immediate functional and performance requirements.

5.3.3 Tactics and Maintainability

For the purposes of this thesis, the most relevant finding is which tactics practitioners associate with maintainability improvements:

Tactic	Effect on Maintainability
Heartbeat	Positive
Time stamp	Positive
Sanity checking	Positive
Functional redundancy	Positive
Analytical redundancy	Positive
Recovery from attacks	Positive
Authentication	Positive
Resource pooling	**Negative** (can hinder maintainability)

This finding is nuanced. Tactics designed primarily for reliability or security (like Heartbeat, Sanity checking) can have **positive side-effects on maintainability** because they promote modular, well-separated components. Conversely, Resource pooling — while excellent for performance — can increase coupling and make the system harder to maintain.

5.3.4 Little-Known Relationships

Perhaps the most surprising finding is that **approximately 21% of the QA-tactic relationships discovered on Stack Overflow are "little-known"** — they are not documented in the academic literature or standard textbooks like Bass et al. [1]. These relationships emerge from real-world experience: practitioners discover through building systems that certain tactics affect quality attributes in ways that formal catalogs do not predict.

> "About 21% of the extracted QA-AT relationships are additional to the ones documented in the literature, which we call 'little-known' design relationships." [22]

5.3.5 Discussion Context

The study also analyzed *how* practitioners discuss tactics. Four main discussion topics emerged:

Topic	Share	Description
Architecture patterns	47%	How to implement a tactic within a given architectural pattern
Design context	28%	When and where to apply a tactic based on project constraints
Design decision evaluation	15%	Trade-offs and comparisons between alternative tactics
AT application in existing systems	11%	How to introduce a tactic into an existing codebase

The dominance of "architecture patterns" (47%) suggests that practitioners think about tactics primarily in terms of their interaction with patterns — reinforcing the pattern-tactic relationship discussed in earlier chapters. The relatively small share of "AT application in existing systems" (11%) is notable: it confirms that **introducing tactics into existing code is rare in practice**, even though it is exactly what is needed to combat erosion.

5.4 The Detection-Remediation Disconnect

5.4.1 Synthesizing the Evidence

The three studies reviewed in this chapter converge on a single, critical insight: **detecting architectural problems is not the same as fixing them**. Let us trace the argument:

1. **Li et al. (2021)** establish that erosion is pervasive, well-documented, and severe. They catalog 35 detection tools and identify 13 categories of causes. Yet they also

report an 82.2% research-practice gap — developers do not use these tools. The study explicitly suggests that "leveraging machine learning to automatically detect AEr symptoms" is a promising direction, but stops short of proposing automated *remediation*.

2. **Rosik et al. (2011)** prove empirically that detection alone fails. They detected 9 violations, presented them to the developers, and **zero were fixed**. The barriers are not ignorance but rather risk, cost, and time. Developers *knew* about the problems and still chose to live with them.

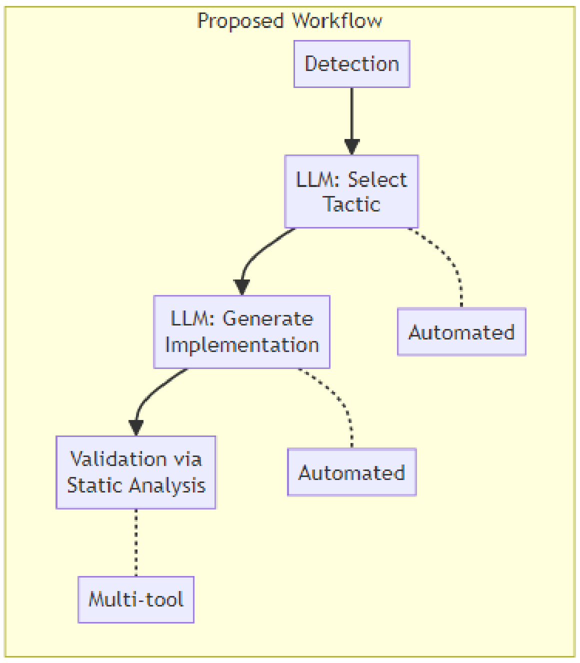
3. **Bi et al. (2021)** show that knowledge about tactics exists in the developer community — 4,195 Stack Overflow posts discuss how tactics relate to quality attributes. But this knowledge is **fragmented, informal, and biased** toward Performance and Security. Maintainability tactics are under-discussed, and only 11% of discussions address applying tactics to existing systems.

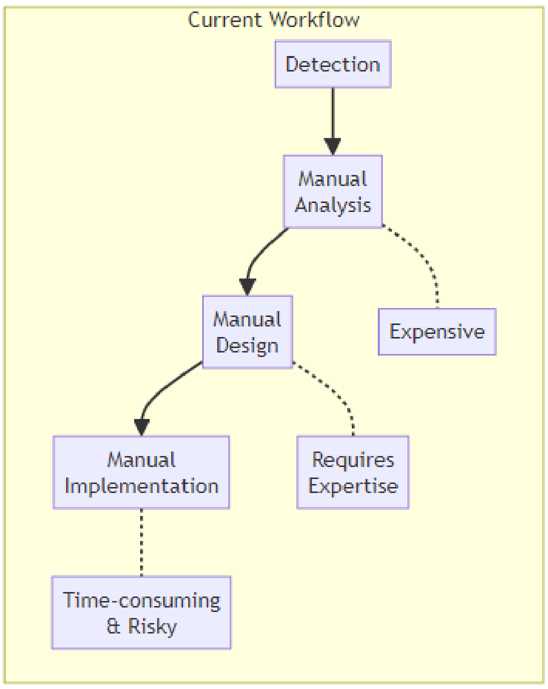
5.4.2 The Gap

The synthesis reveals a **three-part gap**:

Gap	Evidence
Detection exists but is underused	35 tools cataloged, but 82.2% of research is academic; developers do not adopt them [4]
Detection does not lead to action	0/9 violations fixed even after explicit identification [8]
Knowledge exists but is fragmented	4,195 SO posts on tactics, but 21% of relationships are undocumented; maintainability is under-discussed [22]

What is missing is the **middle step** — a mechanism that takes detected problems and implements solutions. The following diagram contrasts the current manual workflow with the envisioned automated approach:





Each manual step is a barrier. Each barrier gives developers a reason to defer the fix. The result is the vicious cycle that Li et al. describe: erosion generates technical debt, which generates more erosion, which further degrades the architecture.

5.4.3 Automated Remediation as the Missing Link

This is where the thesis contribution enters. The central research question is not "can we detect erosion?" (we can) or "do tactics exist to address it?" (they do) but rather: **can an LLM bridge the gap between detection and implementation?**

If an LLM can reliably (a) analyze a detected violation, (b) select an appropriate architectural tactic from a catalog, and (c) implement that tactic in the existing codebase while preserving behavior, then the three barriers identified by Rosik et al. — risk, cost, and time — are substantially reduced:

- **Risk** is reduced because the LLM's changes can be validated by static analysis and testing before deployment.
- **Cost** is reduced because no human architect needs to design the solution.
- **Time** is reduced because implementation is automated — it can run in a CI/CD pipeline on every commit, addressing Rosik et al.'s recommendation for continuous monitoring.

The remaining chapters of this guide explore whether this vision is achievable. Chapter 6 examines how we measure whether the LLM's changes actually improve maintainability. Chapters 7 and 8 examine the LLM capabilities and limitations that determine the feasibility of automated tactic implementation.

5.5 Review Questions

1. **Define** architecture erosion and architecture drift. Give one example of each in the context of a layered web application.
2. Li et al. identify four perspectives on architecture erosion. **List** all four and **explain** why the "quality degradation" perspective is particularly relevant for long-lived systems.
3. The IBM Dublin case study found that 0 out of 9 detected violations were fixed. **Identify** three barriers to remediation from the developer interviews and **explain** how each one could be addressed by an automated approach.
4. Bi et al. found that 21% of QA-tactic relationships on Stack Overflow are "little-known." **Discuss** what this implies about the completeness of formal tactic catalogs and its consequences for automated tactic selection.
5. **Draw** the detection-remediation disconnect as a diagram. Label each gap and identify which gap an LLM-based approach would address.
6. Rosik et al. recommend continuous monitoring over batch detection. **Explain** why, using the concept of *accumulated dependencies* on top of violations.
7. Li et al. report that 57.1% of erosion detection tools are Java-focused. **Discuss** the implications of this language bias for projects written in Python, JavaScript, or Go.

Chapter 6

Maintainability Assessment Methods

Learning objectives. After reading this chapter you will be able to (1) calculate the Maintainability Index, Cyclomatic Complexity, and Halstead metrics for a given code sample, (2) explain the Chidamber-Kemerer object-oriented metric suite and its relevance to coupling and cohesion, (3) use Radon to compute metrics for Python code, (4) describe the agreement crisis among static analysis tools and its implications for research, (5) explain the SQALE model and SonarQube's technical debt calculation, (6) design a multi-tool validation strategy for before/after maintainability comparisons, and (7) evaluate the strengths and limitations of current maintainability metric frameworks.

6.1 Maintainability Metrics

Maintainability is one of the eight quality characteristics defined by ISO/IEC 25010. But unlike functional correctness, it cannot be directly observed — it must be *measured* through proxy metrics. This section presents the major metric families used in software engineering research, with formulas, interpretations, and worked examples.

Ardito et al. [23] conducted a systematic literature review of 43 studies (2000–2019), cataloging **174 distinct maintainability metrics**. However, 75% of those metrics appear in only a single paper. The field has converged on a much smaller set of widely-adopted metrics, which we cover here.

6.1.1 Maintainability Index (MI)

The Maintainability Index is a composite metric designed to provide a single number summarizing how easy a piece of code is to maintain. It was originally proposed by Oman and Hagemeister [24] and later adopted by Microsoft Visual Studio and the Python tool Radon.

Formula (3-metric version):

$$MI = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(LOC)$$

Where:

Symbol	Meaning
V	Halstead Volume (see Section 6.1.3)
G	Cyclomatic Complexity (see Section 6.1.2)
LOC	Lines of Code (logical, non-comment)

Formula (4-metric version, with comments):

$$MI = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(LOC) + 50 \cdot \sin(\sqrt{2.4 \cdot perCM})$$

Where $perCM$ is the percentage of comment lines. The comment term rewards well-documented code.

Interpretation:

MI Range	Rating	Meaning
> 85	A	Easy to maintain
$65 - 85$	B	Moderately maintainable
< 65	C	Difficult to maintain

Controversy. The MI is widely used but not universally accepted. Ardito et al. [23] note that Ostberg and Wagner express doubts about MI's effectiveness, while Sarwar et al. find it efficient. The main criticism is that MI collapses multiple dimensions into a single number, hiding important distinctions. A function with high complexity but low volume might receive the same MI score as one with low complexity but high volume — yet these require very different remediation strategies.

Practical note. Radon (Python) computes MI per module. Microsoft Visual Studio computes it per method and per class. The exact formula varies slightly between implementations, so always report which tool and version you used.

6.1.2 Cyclomatic Complexity (CC)

Cyclomatic Complexity, introduced by McCabe [25], measures the number of linearly independent paths through a function's control flow graph. It is the most widely used single metric for code complexity.

Formula:

$$M = E - N + 2P$$

Where:

Symbol	Meaning
E	Number of edges in the control flow graph
N	Number of nodes in the control flow graph

Symbol	Meaning
P	Number of connected components (usually 1 for a single function)

Practical shortcut. For a single function, CC equals the number of decision points plus one. Each `if`, `elif`, `for`, `while`, `and`, `or`, `except`, and `case` adds one to the count.

Interpretation:

CC Range	Risk Level	Meaning
1 – 10	Low	Simple, well-structured, easy to test
11 – 20	Moderate	More complex, higher testing effort
21 – 50	High	Complex, difficult to test and maintain
> 50	Very high	Untestable, error-prone, should be refactored

Visser [16] recommends a stricter threshold of **CC <= 5** for individual units of code, based on SIG's benchmark of hundreds of real systems. This stricter threshold aligns with the guideline "Write Simple Units of Code."

Example: Calculating CC step by step.

Consider this Python function:

```python
def categorize_score(score, bonus=False):
    """Categorize a student's score into a grade."""
    if score < 0 or score > 100:         # Decision 1 (if), Decision 2 (or)
        raise ValueError("Invalid score")

    if bonus:                             # Decision 3
        score = min(score + 5, 100)

    if score >= 90:                       # Decision 4
        return "A"
    elif score >= 80:                     # Decision 5
        return "B"
    elif score >= 70:                     # Decision 6
        return "C"
    elif score >= 60:                     # Decision 7
        return "D"
    else:
        return "F"
```

Counting decision points:

Decision Point	Statement	Count
1	`if score < 0`	+1
2	`or score > 100`	+1
3	`if bonus`	+1

Decision Point	Statement	Count
4	`if score >= 90`	+1
5	`elif score >= 80`	+1
6	`elif score >= 70`	+1
7	`elif score >= 60`	+1

CC = 7 + 1 (base) = 8. This falls in the "low risk" range (1–10) but exceeds Visser's stricter threshold of 5. Reducing it would require extracting the grade thresholds into a data structure or splitting the validation from the categorization.

6.1.3 Halstead Metrics

Halstead's Software Science metrics [26] treat a program as a sequence of operators and operands and derive complexity measures from their counts. While sometimes criticized as overly reductive, Halstead metrics remain widely used because they capture aspects of code density and cognitive effort that CC alone misses.

Base counts:

Symbol	Meaning	Example
n_1	Number of **distinct** operators	`+`, `=`, `if`, `return`, `def`, `()`, etc.
n_2	Number of **distinct** operands	Variable names, constants, string literals
N_1	**Total** number of operator occurrences	Every time an operator is used
N_2	**Total** number of operand occurrences	Every time an operand is used

Derived metrics:

Metric	Formula	Interpretation
Program Length	$N = N_1 + N_2$	Total token count
Vocabulary	$n = n_1 + n_2$	Number of distinct tokens
Volume	$V = N \cdot \log_2(n)$	Information content of the program
Difficulty	$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$	How hard the program is to write or understand
Effort	$E = D \cdot V$	Total cognitive effort to develop the program
Time to implement	$T = \frac{E}{18}$	Estimated time in seconds (Stroud number = 18)
Bugs predicted	$B = \frac{V}{3000}$	Estimated number of bugs delivered

Volume is the most commonly used Halstead metric in maintainability research. It appears directly in the MI formula. Higher volume means more information content, which generally correlates with harder-to-maintain code.

Difficulty captures how error-prone the code is. It increases when operators are diverse (high n_1) and when operands are reused frequently (high N_2/n_2).

6.1.4 Chidamber-Kemerer (CK) Object-Oriented Suite

For object-oriented systems, the CK metrics suite [27] provides measures of coupling, cohesion, and inheritance complexity at the class level. These metrics are especially relevant for architectural tactics, which often involve restructuring class relationships.

Metric	Full Name	Definition	What It Measures
CBO	Coupling Between Objects	Count of classes to which a given class is coupled (uses methods, accesses variables, inherits, etc.)	Inter-class coupling
RFC	Response For a Class	Count of methods that can potentially be executed in response to a message received by an object of the class (own methods + directly called methods)	Communication complexity
WMC	Weighted Methods per Class	Sum of cyclomatic complexities of all methods in the class	Class complexity
DIT	Depth of Inheritance Tree	Maximum length from the class to the root of the inheritance hierarchy	Inheritance complexity
NOC	Number of Children	Count of immediate subclasses	Inheritance breadth
LCOM	Lack of Cohesion in Methods	Number of pairs of methods that do not share instance variables minus the number that do (if negative, set to 0)	Internal cohesion (inversely)

Relevance to architecture:

- **CBO** is the most frequently cited maintainability metric in the literature — 18 mentions across the 43 studies in Ardito et al.'s SLR [23]. High CBO means a class is tightly coupled to many others, making changes ripple across the codebase. Architectural tactics that **reduce coupling** (encapsulation, intermediaries, restrict dependencies) directly target CBO.
- **RFC** is the second most cited (17 mentions). A class with high RFC has a large "blast radius" — calling one of its methods can trigger a chain of calls across many classes. This makes the system harder to understand, test, and modify.
- **LCOM** measures cohesion inversely. A high LCOM means the class's methods operate on disjoint sets of instance variables, suggesting the class has multiple unrelated responsibilities. This is often a sign that the class should be split — a tactic known as **increase cohesion**.
- **WMC** directly reflects internal complexity. Visser [16] maps this to the guideline "Write Simple Units of Code" and recommends keeping individual method CC <= 5, which in turn keeps WMC manageable.

6.1.5 Comprehensive Metric Reference Table

Metric	Formula / Definition	Level	Tool(s)	Threshold	Source
MI	$171 - 5.2\ln(V) - 0.23G - 16.2\ln(LOC)$	Module	Radon, VS	>85 good, <65 bad	[24]
CC	$E - N + 2P$ (decision points + 1)	Function	Radon, PMD, SonarQube	1-10 low risk	[25]
Halstead Volume	$V = N \cdot \log_2(n)$	Function	Radon, JHawk	Lower is simpler	[26]
Halstead Difficulty	$D = (n_1/2) \cdot (N_2/n_2)$	Function	Radon, JHawk	Lower is simpler	[26]
Halstead Effort	$E = D \cdot V$	Function	Radon, JHawk	Lower is simpler	[26]
LOC	Logical lines of code	Function/Module	All tools	Minimize	Various
CBO	Classes coupled to	Class	CKJM, SonarQube	<9 recommended	[27]
RFC	Methods invocable on message	Class	CKJM, Understand	Lower is better	[27]
WMC	Sum of method CCs	Class	CKJM, SonarQube	Lower is better	[27]
DIT	Max inheritance depth	Class	CKJM, Understand	<6 recommended	[27]
NOC	Immediate subclass count	Class	CKJM	Context-dependent	[27]

Metric	Formula / Definition	Level	Tool(s)	Threshold	Source
LCOM	Cohesion lack (method pairs)	Class	CKJM	0 = perfect cohesion	[27]
TDR	Remediation cost / dev cost	Project	SonarQube	<5% = A rating	SQALE model

6.1.6 Practical Example: Full Metric Calculation with Radon

Consider the following Python module that manages a simple inventory:

```python
# inventory.py
class Inventory:
    """Manages product inventory with basic CRUD operations."""

    def __init__(self):
        self.products = {}
        self.low_stock_threshold = 10

    def add_product(self, name, quantity, price):
        if name in self.products:
            self.products[name]["quantity"] += quantity
        else:
            self.products[name] = {
                "quantity": quantity,
                "price": price
            }

    def remove_product(self, name):
        if name not in self.products:
            raise KeyError(f"Product '{name}' not found")
        del self.products[name]

    def get_total_value(self):
        total = 0
        for name, info in self.products.items():
            total += info["quantity"] * info["price"]
        return total

    def get_low_stock_items(self):
        low_stock = []
        for name, info in self.products.items():
            if info["quantity"] < self.low_stock_threshold:
                low_stock.append(name)
        return low_stock

    def apply_discount(self, name, percent):
        if name not in self.products:
            raise KeyError(f"Product '{name}' not found")
        if percent < 0 or percent > 100:
            raise ValueError("Discount must be between 0 and 100")
        factor = 1 - (percent / 100)
        self.products[name]["price"] *= factor
```

Computing metrics with Radon:

```
# Maintainability Index (per module)
$ radon mi inventory.py -s
inventory.py - A (67.75)

# Cyclomatic Complexity (per function/method)
$ radon cc inventory.py -s
inventory.py
    C 3:0 Inventory - B (11)
        M 7:4 __init__ - A (1)
        M 10:4 add_product - A (2)
        M 18:4 remove_product - A (2)
        M 23:4 get_total_value - A (1)
        M 29:4 get_low_stock_items - A (2)
        M 35:4 apply_discount - A (3)

# Halstead metrics (per function)
$ radon hal inventory.py
inventory.py:
    __init__:
        h1: 2     h2: 3     N1: 2     N2: 3
        vocabulary: 5     length: 5
        volume: 11.61     difficulty: 1.00
        effort: 11.61     time: 0.65     bugs: 0.00
    add_product:
        h1: 7     h2: 8     N1: 12    N2: 14
        vocabulary: 15    length: 26
        volume: 101.59    difficulty: 6.13
        effort: 622.47    time: 34.58    bugs: 0.03
    ...
```

Reading the output:

- The module receives an MI of **67.75**, which is a **B rating** (moderately maintainable). This makes sense — the code is straightforward but not trivially simple.
- The class `Inventory` has a total CC of **11** (sum of all method CCs), putting it at risk level B. Individual methods range from CC 1 (`__init__`, `get_total_value`) to CC 3 (`apply_discount`).
- Halstead volume for `add_product` is 101.59, reflecting its use of dictionary operations, conditionals, and multiple parameters.

What these metrics tell us: The `Inventory` class is reasonably maintainable at the method level (all methods have low CC) but is accumulating coupling between responsibilities — it handles CRUD, valuation, stock monitoring, and pricing. An architectural tactic like **Increase Cohesion** (splitting into separate classes for stock management, pricing, and reporting) would reduce WMC and potentially improve MI.

6.2 Static Analysis Tools

Static analysis tools (SATs) automate the computation of code metrics and the detection of quality issues. They analyze source code without executing it, making them fast, repeatable, and suitable for CI/CD integration.

6.2.1 Tool Landscape

Tool	Language(s)	Focus	Open Source	Key Metrics
SonarQube	30+ languages	Technical debt, code smells, bugs, vulnerabilities	Community edition: Yes	SQALE TDR, issue counts, rating (A–E)
PMD	Java, Apex, JS, XML	Rule-based pattern detection, copy-paste detection	Yes	Rule violations, CPD duplicates
CheckStyle	Java	Coding standards, formatting, naming conventions	Yes	Style violations
FindBugs/SpotBugs	Java (bytecode)	Bug patterns in compiled code	Yes (SpotBugs fork)	Bug pattern counts by category
Radon	Python	CC, MI, Halstead, LOC, raw metrics	Yes	CC, MI, Halstead Volume/Difficulty/Effort

Tool	Language(s)	Focus	Open Source	Key Metrics
Pylint	Python	Coding standards, error detection, refactoring suggestions	Yes	Convention/refactor/warning/erro counts, score
ESLint	JavaScript/TypeScript	Pluggable linting, code quality rules	Yes	Rule violations by severity

Each tool occupies a different niche. SonarQube is the broadest in language support and issue taxonomy. Radon is the standard for Python-specific complexity metrics. PMD and CheckStyle focus on Java with different emphases (patterns vs. style). FindBugs/SpotBugs analyzes compiled bytecode, catching issues that source-level tools miss.

6.2.2 The Agreement Crisis

If multiple tools exist for the same purpose, a natural question is: **do they agree?** The answer, based on rigorous empirical evidence, is deeply troubling.

Lenarduzzi et al. [28] conducted the largest empirical comparison of static analysis tools to date. They applied six tools (SonarQube, Better Code Hub, Coverity Scan, FindBugs, PMD, and CheckStyle) to **47 Java projects** from the Qualitas Corpus (a standard benchmark dataset) and measured three things: what each tool detects, how much they agree, and how precise their detections are.

Detection agreement:

Tool Pair	Class-Level Agreement
FindBugs – PMD	9.378% (best)
SonarQube – FindBugs	4.216%
SonarQube – PMD	2.881%
SonarQube – CheckStyle	1.527%
CheckStyle – FindBugs	0.892%
CheckStyle – PMD	0.144% (worst)
Overall (all 6 tools)	**$< 0.4\%$**

Less than 0.4% overall agreement. This means that if you show the same Java class to all six tools, there is less than a 0.4% chance they will all flag the same issue. Even the best pair — FindBugs and PMD — agrees less than 10% of the time.

> "The key results show little to no agreement among the tools and a low degree of precision." [28]

Why do tools disagree? They operate at different abstraction levels (source code vs. bytecode), use different rule sets (936 unique rules across all six tools), and have different goals (style enforcement vs. bug detection vs. debt estimation). Even when two tools have rules that *sound* the same, only 6 out of 66 candidate rule pairs were found to detect the same underlying issue.

Precision:

Tool	Precision	True Positives / Sample
CheckStyle	86%	330 / 384
FindBugs	57%	217 / 379
PMD	52%	199 / 380
Coverity Scan	37%	136 / 367
Better Code Hub	29%	109 / 375
SonarQube	18%	69 / 384

SonarQube has the **lowest precision at 18%** — meaning that roughly 4 out of 5 issues it flags are false positives. However, this must be interpreted carefully: SonarQube also has the broadest rule set and detects the widest variety of issue types. CheckStyle has the highest precision (86%) but predominantly detects formatting and naming violations, not design or architectural issues.

> "There is no silver bullet that is able to guarantee source code quality assessment on its own." [28]

Implication for research: Any study that uses a single static analysis tool to measure maintainability improvement is measuring what that *particular tool* thinks, not an objective ground truth. Reported improvements may reflect tool-specific biases rather than genuine quality changes.

6.3 False Positives and False Negatives

The agreement crisis concerns **what tools detect**. An equally important question is **what they get wrong** and **what they miss entirely**.

6.3.1 Root Causes of Errors

Cui et al. [29] conducted the first systematic study of false negatives (FNs) and false positives (FPs) using the analyzers' own bug repositories. They examined **350 confirmed, developer-fixed FN/FP issues** across PMD, SpotBugs, and SonarQube.

Root Cause	Frequency	Description
Missing cases	35.7%	Analyzer lacks handling for specific code patterns (e.g., missing whitelisted items, unhandled edge cases)

Root Cause	Frequency	Description
Unhandled language features	28.9%	New Java features (lambdas, records, pattern matching) not supported by analysis rules
Analysis module errors	Varies	Bugs in type resolution, data-flow analysis, or symbolic execution engines
Incorrect rule logic	Varies	The rule's implementation does not match its specification

6.3.2 What Triggers Failures

The study identified **10 input characteristics** that commonly trigger FN/FP errors:

Input Characteristic	Frequency	Example
Annotations	21.07%	Tools fail to model annotation semantics (e.g., `@Nullable`, `@Override`)
Complex expressions	15.4%	Nested ternary operators, lambda chains
Generics / Type parameters	12.1%	Parameterized types confuse type resolution
Try-with-resources	8.2%	Resource management constructs misanalyzed
Inter-procedural flows	Varies	Tools cannot track values across method boundaries

The false negative problem is especially concerning. While false positives waste developer time, false negatives provide a *false sense of security*. If a tool reports no issues, developers may believe the code is clean when it is not. Cui et al. found that PMD's data-flow analysis is limited to reaching-definition analysis only and cannot handle basic inter-procedural call flows. SonarQube's symbolic execution engine has limitations in tracking runtime types outside method boundaries.

> "Unfortunately, this rule is (implemented as) AST-based, and it brings some limitations. So to eliminate these false positives the rule should rely on the data flow." — SonarQube developer [29]

6.3.3 Implications for Before/After Studies

When a thesis compares maintainability metrics before and after an intervention (such as applying an architectural tactic), the reliability of the measurement instrument matters enormously. If the tool has:

- **High FP rate:** Improvements may appear where none exist (the tool stops flagging false positives after refactoring changes the code structure).
- **High FN rate:** Real problems may be hidden both before and after, making the comparison meaningless.
- **Inconsistent behavior:** The same pattern in different syntactic forms may be detected before refactoring but missed after (or vice versa).

This is why multi-tool validation (Section 6.5) and statistical testing are essential.

6.4 SonarQube and the SQALE Model

Despite its low precision, SonarQube is the most widely used static analysis platform in both industry and research. Understanding its measurement model is essential for interpreting its output correctly.

6.4.1 The SQALE Model

SonarQube uses the **SQALE** (Software Quality Assessment based on Lifecycle Expectations) model to estimate technical debt. The core idea is:

1. For every detected issue, SonarQube estimates the **remediation effort** — how many minutes it would take a developer to fix the issue.
2. The sum of all remediation efforts is the **technical debt** (expressed in minutes, hours, or days).
3. The **Technical Debt Ratio (TDR)** normalizes this against the estimated development cost:

$$TDR = \frac{\text{Remediation Cost (total fix effort)}}{\text{Development Cost (estimated effort to rewrite)}}$$

4. TDR maps to a **letter rating**:

Rating	TDR Range	Interpretation
A	$< 5\%$	Minimal technical debt
B	6 – 10%	Low technical debt
C	11 – 20%	Moderate technical debt
D	21 – 50%	High technical debt
E	$> 50\%$	Very high technical debt, remediation cost exceeds half the rewrite cost

6.4.2 Issue Taxonomy

SonarQube classifies issues into three categories:

Category	Definition	Example
Code Smells	Maintainability issues that make code harder to understand or modify	Long methods, excessive parameters, god classes
Bugs	Reliability issues that could cause incorrect behavior	Null pointer dereferences, resource leaks
Vulnerabilities	Security issues that could be exploited	SQL injection, hardcoded credentials

6.4.3 SonarQube in Practice

Nocera et al. [30] mined **321 GitHub projects** using SonarQube Cloud to understand how the tool is actually configured and used in open-source development.

Key findings:

Finding	Value
Projects using default "Sonar way" quality gate	55%
Projects with customized quality gates	45%
Projects enforcing maintainability rating on new code	89.06%
Projects enforcing security rating on new code	90.94%
Projects enforcing reliability rating on new code	88.68%
Projects enforcing coverage on new code	77.36% (least common built-in condition)

The finding that **89% of projects enforce a maintainability rating** on new code validates that maintainability is a priority quality attribute in real-world development, not just an academic concern. This also means SonarQube's maintainability rating is the *de facto* industry standard for automated maintainability assessment.

Important caveat from Nocera et al.: Issue count reduction does not necessarily equal debt reduction. They found ratios as low as 0.71 between the two — meaning you can fix many issues without significantly reducing overall technical debt if the remaining issues are high-effort ones. This has direct implications for evaluating LLM-driven interventions: counting resolved issues is not sufficient; you must track actual debt reduction (in remediation-time units).

6.4.4 Cloud vs. On-Premise

Nocera et al. also note differences between SonarQube Cloud and on-premise installations. The cloud version follows the "Clean as You Code" philosophy, which focuses quality gates on *new code* rather than the entire codebase. This means the quality gate only triggers on changes since the last version, which is philosophically aligned with the thesis approach:

we apply tactics to specific parts of the code and measure whether those changes improve metrics.

6.5 Multi-Tool Validation Strategy

Given the agreement crisis ($< 0.4\%$) and the precision problems documented above, relying on a single tool for maintainability assessment is methodologically unsound. This section presents a principled strategy for combining multiple tools.

6.5.1 Why One Tool Is Never Enough

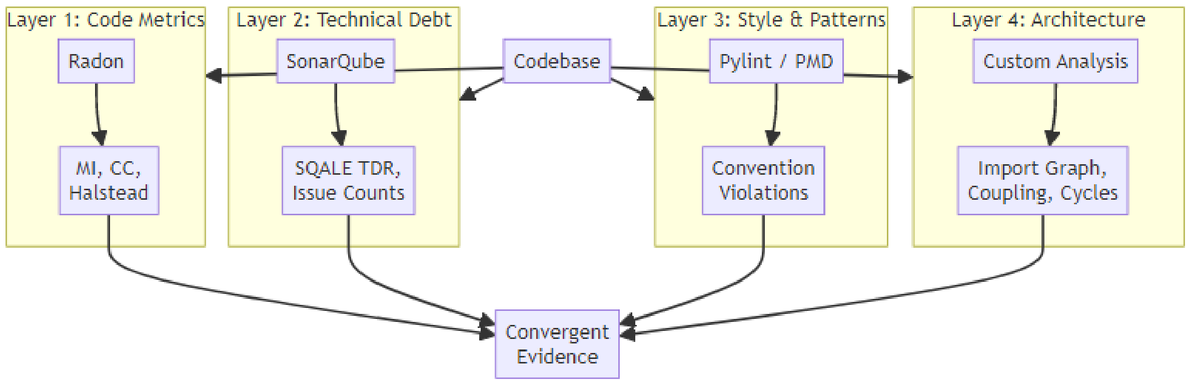
The evidence is unambiguous:

Problem	Evidence	Source
Tools disagree on what to flag	$< 0.4\%$ overall agreement	[28]
Tools have high false positive rates	SonarQube: 18%, PMD: 52% precision	[28]
Tools have systematic blind spots	35.7% of FN/FP issues from missing cases	[29]
Tools miss entire categories of issues	FN rates significant across all analyzers	[29]
Issue counts and debt reduction diverge	Ratio as low as 0.71	[30]

No single tool provides a complete picture. Different tools catch different things. Combining them increases both coverage and confidence.

6.5.2 Recommended Tool Combination

For a research study evaluating maintainability before and after an intervention, the following four-layer strategy is recommended:



Layer	Tool	What It Measures	Why Include It
1. Code-level metrics	Radon (Python)	MI, CC, Halstead Volume/Difficulty/Effort, LOC, raw metrics	Precise, reproducible, formula-based; no subjective rule interpretation
2. Technical debt estimation	SonarQube	SQALE TDR, issue counts by category (smells, bugs, vulnerabilities), maintainability rating	Industry standard; normalized debt metric enables cross-project comparison
3. Style and pattern violations	Pylint (Python) or PMD (Java)	Convention violations, refactoring suggestions, error patterns, code score	Catches language-specific anti-patterns that Radon misses
4. Architecture-level	Custom analysis	Import graph coupling, module cohesion, layer violations, dependency cycles	Captures architecture-level effects that file-level tools cannot detect

Layer 1 (Radon) provides the *objective baseline*: formulas computed deterministically from the source code. There is no ambiguity about what CC or MI means. These metrics are the primary comparison variables.

Layer 2 (SonarQube) provides the *industry-standard view*: technical debt in minutes, severity-classified issues, and the A–E rating that practitioners and CI/CD pipelines actually use. Despite its low precision (18%), it captures the broadest range of issue types and is the metric most likely to be meaningful to practitioners.

Layer 3 (Pylint/PMD) provides *language-specific detail*: naming conventions, import order, unused variables, and refactoring opportunities that neither Radon nor SonarQube focus on. Pylint's numeric score (0–10) provides a simple before/after comparison point.

Layer 4 (Custom) is essential because **no commercial tool measures architecture-level maintainability well**. Import graph analysis can detect coupling between modules, identify circular dependencies, and measure whether tactic implementations actually change the dependency structure. This is the layer most directly relevant to architectural tactics.

6.5.3 Statistical Methods for Before/After Comparison

When comparing metrics before and after applying an architectural tactic, raw numbers are insufficient. You need statistical evidence that the change is both *significant* (not due to chance) and *meaningful* (large enough to matter).

Effect size with Cliff's Delta (δ):

Cliff's Delta is a non-parametric effect size measure suitable for ordinal data and non-normal distributions. It quantifies how often values in one group exceed values in the other.

$\lvert\delta\rvert$ Range	Interpretation
≤ 0.147	Negligible effect
$0.148 - 0.330$	Small effect
$0.331 - 0.474$	Medium effect
> 0.474	Large effect

Significance testing with Wilcoxon signed-rank test:

When the same modules are measured before and after treatment (a paired design), the Wilcoxon signed-rank test is appropriate because it does not assume normality. A p-value < 0.05 indicates a statistically significant change.

Multiple comparison correction with Benjamini-Hochberg:

When testing multiple metrics simultaneously (e.g., CC, MI, Halstead, and CBO for the same dataset), the risk of false positives increases. The Benjamini-Hochberg FDR (False Discovery Rate) correction controls the expected proportion of false discoveries. This is preferable to the more conservative Bonferroni correction when testing many hypotheses.

Example analysis workflow:

```python
from scipy.stats import wilcoxon
from cliffs_delta import cliffs_delta  # pip install cliffs-delta

# Before and after MI values for 50 modules
mi_before = [72.3, 65.1, 81.4, ...]  # measured before tactic
mi_after  = [78.9, 71.2, 83.1, ...]  # measured after tactic

# 1. Effect size
delta, interpretation = cliffs_delta(mi_after, mi_before)
print(f"Cliff's Delta: {delta:.3f} ({interpretation})")

# 2. Significance test
stat, p_value = wilcoxon(mi_before, mi_after, alternative='less')
print(f"Wilcoxon p-value: {p_value:.4f}")

# 3. Apply Benjamini-Hochberg correction if testing multiple metrics
from statsmodels.stats.multitest import multipletests
p_values = [p_mi, p_cc, p_halstead, p_cbo]  # one per metric
rejected, corrected_p, _, _ = multipletests(p_values, method='fdr_bh')
```

This workflow ensures that reported improvements are statistically rigorous and not artifacts of tool noise or multiple testing.

6.6 Ardito et al.'s SLR on Maintainability Metrics

We have referenced Ardito et al. [23] throughout this chapter, but their systematic literature review deserves dedicated attention as the most comprehensive survey of the metric landscape.

6.6.1 Scope and Method

The SLR followed Kitchenham's guidelines, screening 801 papers from four digital libraries (ACM, IEEE Xplore, Scopus, Web of Science) and selecting 43 primary studies published between 2000 and 2019.

6.6.2 The 174 Metrics Problem

The SLR identified **174 distinct maintainability metrics**, organized into 10 metric suites. However, the distribution is extremely long-tailed:

- **75% of all 174 metrics are mentioned by only a single paper.** There is no broad consensus on most of these metrics.
- The field has effectively converged on a core set of **15 most-mentioned metrics** (those scoring above the median in Ardito et al.'s frequency analysis).

The 15 most-mentioned metrics:

Rank	Metric	Mentions	Suite
1	CBO (Coupling Between Objects)	18	CK
2	RFC (Response For a Class)	17	CK
3	LOC (Lines of Code)	15	Size
4	CC (Cyclomatic Complexity)	14	McCabe
5	WMC (Weighted Methods per Class)	14	CK
6	DIT (Depth of Inheritance Tree)	13	CK
7	LCOM (Lack of Cohesion in Methods)	12	CK
8	NOC (Number of Children)	11	CK
9	Halstead suite (6 metrics)	10	Halstead
10	MI (Maintainability Index)	9	Composite
11	NOM (Number of Methods)	8	Size
12	NPM (Number of Public Methods)	7	CK Extended
13	MPC (Message Passing Coupling)	6	Coupling
14	CE (Efferent Coupling)	6	Martin
15	CLOC (Comment Lines of Code)	5	Size

Notice that **5 of the top 8 metrics are from the CK suite**. Object-oriented coupling and cohesion metrics dominate the maintainability measurement landscape.

6.6.3 Tool Coverage

Ardito et al. catalog **19 available tools** (6 closed-source, 13 open-source) across 34 programming languages. Key findings about tool coverage:

- **CC** is supported by 13 of 19 tools — the best-supported metric.
- **LOC** is supported by 14 of 19 tools.
- **LCOM2 and MPC** are not explicitly supported by any open-source tool, creating a coverage gap for researchers who need fully open-source toolchains.

- **Java** has the best tool support (13 tools), followed by C/C++ (12), JavaScript (10), and C# (9).
- For Java, the optimal tool set requires **5 tools** to cover all 15 most-mentioned metrics.
- **CKJM** (Chidamber-Kemerer Java Metrics) is the most cited tool in the literature (5 papers) but supports only Java.

6.6.4 The Standardization Gap

Perhaps the most important finding for researchers is the **lack of standardization**. Different tools compute "the same" metric differently:

- **LOC** has at least 3 common definitions: physical lines, logical lines (statements), and non-blank non-comment lines. Different tools use different definitions.
- **MI** has at least 2 formulas in common use (3-metric and 4-metric versions), and Radon normalizes it to a 0–100 scale while some tools use the raw formula that can exceed 100.
- **LCOM** has multiple definitions (LCOM1 through LCOM5), and tools rarely specify which version they compute.

This means that metric values from different tools are **not directly comparable**. Research studies must report the exact tool, version, and configuration used, and comparisons between studies that used different tools must be interpreted with extreme caution.

6.6.5 The Missing Benchmark

Ardito et al. identify a critical gap: **there is no standard benchmark dataset for calibrating maintainability tools**. Unlike in other areas of software engineering (e.g., the Defects4J benchmark for fault localization), there is no agreed-upon codebase with known maintainability scores that can be used to validate tools. This means every tool's thresholds are calibrated against different data, making cross-tool comparison inherently difficult.

Visser [16] partially addresses this through the SIG benchmark, which rates systems on a 1–5 star scale based on empirical benchmarks from hundreds of real systems. SIG recalibrates annually, ensuring thresholds reflect current coding practices. The predictive power is demonstrated by their finding that issue resolution is **2x faster in 4-star vs. 2-star systems** — a concrete validation that the metrics predict actual maintainability outcomes.

However, the SIG benchmark is proprietary and language-specific (Java-focused), limiting its availability to the broader research community.

6.7 Review Questions

1. **Calculate** the Maintainability Index for a module with Halstead Volume = 500, Cyclomatic Complexity = 12, and LOC = 80. What rating does it receive?

2. A function has the following control flow: one `if-elif-else` chain with 4 branches, a `for` loop, and a `try-except` block. **Calculate** its Cyclomatic Complexity. Is this acceptable by McCabe's threshold? By Visser's threshold?
3. **Explain** the difference between Halstead Volume and Halstead Difficulty. Why does the MI formula use Volume rather than Difficulty?
4. Lenarduzzi et al. found that SonarQube has 18% precision. **Discuss** what this means for a researcher who reports "SonarQube detected 200 fewer code smells after applying our technique." How confident should we be in this claim?
5. **Design** a multi-tool validation strategy for evaluating the maintainability impact of refactoring a Python web application. Specify which tools you would use, what metrics you would track, and how you would determine if the improvement is statistically significant.
6. Ardito et al. found that 75% of the 174 cataloged metrics appear in only a single paper. **Discuss** what this implies about the maturity of maintainability measurement as a field. Should researchers use only the top 15 metrics, or is there value in exploring less-popular ones?
7. Explain the SQALE Technical Debt Ratio. A project has a remediation cost of 40 hours and an estimated development cost of 500 hours. **Calculate** the TDR and assign a SonarQube rating.
8. Cui et al. found that "missing cases" account for 35.7% of FN/FP issues. **Give** a concrete example of how this could affect a before/after maintainability study (i.e., how a missing case could make a refactoring appear to improve or worsen a metric when it actually did neither).
9. Nocera et al. found that issue count reduction and debt reduction can diverge (ratio 0.71). **Explain** why this happens and what metric (issue count or TDR) is more meaningful for evaluating architectural tactic effectiveness.
10. **Compare** the SIG assessment model (Visser) with the SQALE model (SonarQube). What are the strengths and weaknesses of each for evaluating LLM-generated code changes?

Chapter 7

Large Language Models for Code Refactoring

Learning objectives. After reading this chapter you should be able to (1) describe the paradigm shift from syntax-level code completion to semantic-level code transformation, (2) evaluate the strengths and limitations of standalone LLM refactoring, (3) design effective prompts for refactoring tasks using empirically validated strategies, (4) explain the multi-agent and tool-integrated pipeline architectures that dramatically improve refactoring success rates, (5) compare behavior preservation methods and their reliability, and (6) articulate the gap between code-level refactoring and the architecture-level transformations that motivate this thesis.

7.1 The LLM Revolution in Software Engineering

7.1.1 From code completion to code transformation

The application of large language models to software engineering has undergone a rapid and consequential evolution. GitHub Copilot, launched in 2021, demonstrated that transformer-based models trained on vast corpora of source code could autocomplete functions and suggest boilerplate with surprising accuracy. But autocomplete is fundamentally a *syntactic* operation: the model predicts the next likely tokens given a local context window.

By 2023, the field had shifted decisively toward *code transformation* — the use of LLMs not merely to extend code but to restructure, refactor, and improve it. This shift matters because refactoring is not about writing new functionality; it is about understanding *existing* code deeply enough to reorganize it without changing what it does. That requires semantic understanding: knowing what a method *means*, which variables carry state, where side effects lurk, and how components interact. Martinez et al., in their systematic literature review, confirm that the LLM-refactoring intersection has become one of the most active areas in software engineering research [31].

The paradigm shift can be stated simply: **LLMs understand code semantics, not**

just syntax. They can identify that two loops perform the same traversal, that a deeply nested conditional could be flattened into a guard clause, or that three scattered database calls could be consolidated into a repository class. This capability makes them plausible candidates for automated refactoring — and potentially for the architectural transformations that this study guide is building toward.

7.1.2 Key models and their characteristics

Different LLM models exhibit different strengths and weaknesses for refactoring tasks. The table below synthesizes findings from multiple empirical studies to characterize the models most commonly used in the literature.

Model	Provider	Parameters	Best At	Limitations	Source
GPT-3.5	OpenAI	175B	Simple refactorings (rename, reformat); behavior preservation (97.2%)	Defaults to readability improvements regardless of target quality attribute; cannot distinguish coupling from cohesion	[32]
GPT-4 / GPT-4o	OpenAI	~1.8T (est.)	Opportunity identification (86.7% recall with optimized prompts); 63.6% of solutions comparable to or better than human experts	Limited to <300 LOC for practical use; 7.4% of solutions introduce semantic bugs	[33]
GPT-4o-mini	OpenAI	Undisclosed	Multi-agent pipelines (MANTRA: 82.8% success); cost-efficient (<$0.10 per refactoring)	Requires orchestration framework; cannot handle Move Method to new classes	[9], [34]

Model	Provider	Parameters	Best At	Limitations	Source
DeepSeek-V3	DeepSeek	671B (MoE)	Benchmark refactoring (100% success with Step-by-Step/Rule-based instructions on 48/61 Fowler types)	Lower compilation rate on real projects (38–51%); hallucinations from insufficient context	[34]
StarCoder2-15B	BigCode	15B	Code smell reduction (44.4% vs. 24.3% for developers); open-source, training data known	Low Pass@1 (28.4%); struggles with cross-class refactorings; single-file scope	[35]
Gemini 1.0 Pro	Google	Undisclosed	Robust across configurations in iterative pipelines; fewer unsafe solutions (6.6%)	Lower recall than GPT-4 for opportunity identification (21.1% vs. 52.2% with type-specified prompts)	[33], [36]
Claude	Anthropic	Undisclosed	Iterative SonarQube pipelines; agentic coding workflows	Only 0.6% of agentic commits in AIDev dataset; limited empirical data on refactoring-specific performance	[36], [37]

Several patterns emerge from this comparison. First, **model size alone does not predict refactoring success** — GPT-4o-mini within the MANTRA framework (82.8%) vastly outperforms raw GPT-4 (8.7% baseline), demonstrating that pipeline architecture matters more than raw model capability. Second, **open-source models** like StarCoder2-15B show strong performance on pattern-based code improvements but lag behind commercial models on complex structural transformations. Third, **all models** struggle with cross-file, architecture-level reasoning — a limitation we will return to in Section 7.8.

7.2 Standalone LLM Refactoring

7.2.1 DePalma et al.: the pioneering quality-attribute study

The first systematic evaluation of LLM refactoring capabilities across multiple quality attributes was conducted by DePalma et al. [32]. Their study tested ChatGPT (GPT-3.5) on 40 Java code segments, asking it to refactor each segment targeting eight distinct quality attributes: Performance, Complexity, Coupling, Cohesion, Design Size, Readability, Reusability, and Understandability. This produced 320 total refactoring trials.

The headline results were encouraging:

- **99.7% refactoring rate** — ChatGPT produced a refactored version in 319 of 320 trials.
- **97.2% behavior preservation** — 311 of 320 refactored segments maintained their original behavior (assessed via ChatGPT self-evaluation).
- **98.8% documentation accuracy** — ChatGPT correctly identified the goal of its own refactoring in nearly every case.

It is important to understand what these numbers *do* and *do not* mean. The quality attributes tested are *not* Fowler refactoring types (Extract Method, Inline Variable, etc.). They are broader quality concerns. DePalma et al. found that regardless of which quality attribute was specified in the prompt, **ChatGPT predominantly performed generic operations**: variable and method renaming to follow Java conventions, code reformatting, and loop simplification. When asked to improve "coupling," it still produced readability improvements. The authors noted:

> "The two primary attributes ChatGPT reported were readability and maintainability which suggests that ChatGPT has a limited ability to refactor code segments to improve more sophisticated quality attributes like cohesion and coupling." [32]

This finding carries a crucial lesson: **LLMs default to superficial improvements** unless the prompt provides very specific quality-attribute keywords. When the prompt included phrases like "quality and performance," ChatGPT triggered more substantive changes. But even then, it struggled to distinguish between architectural quality attributes that require structural understanding.

7.2.2 Haindl and Weinberger: novice programmer code quality

A complementary perspective comes from Haindl and Weinberger's controlled experiment comparing code written by novice programmers with and without ChatGPT assistance [6]. In a university Java course, 22 students used ChatGPT while 16 did not. Static analysis revealed that the ChatGPT-assisted group produced code with:

- **Significantly lower cyclomatic complexity** ($p < 0.005$)
- **Fewer coding convention violations** across 10 Checkstyle rule categories

However, not all metrics improved. Code length *increased* in the ChatGPT-assisted group, and some SonarQube metrics worsened. This mixed result — improvement on some dimensions, regression on others — foreshadows a theme that will recur throughout this chapter: **refactoring benefits are multi-dimensional**, and a single metric never tells the whole story.

7.2.3 What LLMs handle well vs. poorly

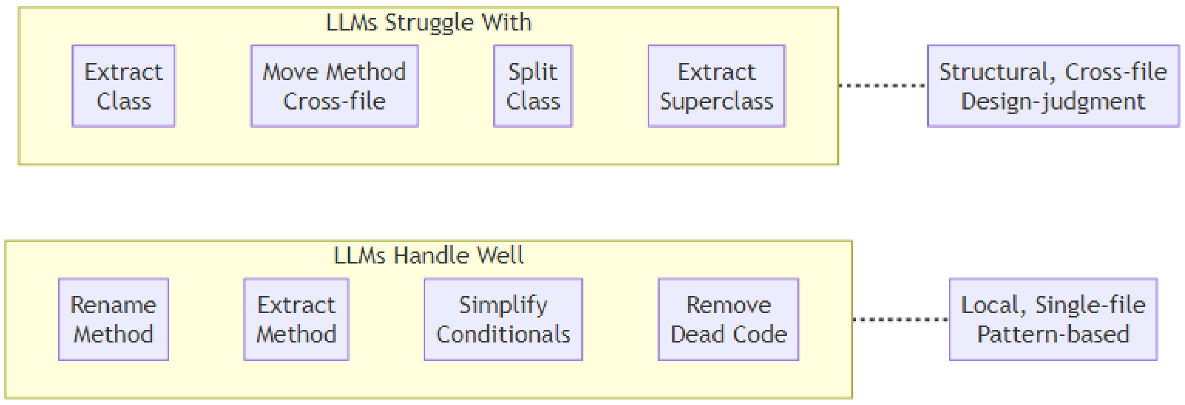
Synthesizing evidence from DePalma et al. [32], Liu et al. [33], Cordeiro et al. [35], and Horikawa et al. [37], we can categorize refactoring types by how well standalone LLMs handle them:

Well Handled	Poorly Handled
Rename Method/Variable	Extract Class
Extract Method	Move Method (cross-file)
Simplify Conditionals	Inline Class
Remove Dead Code	Pull Up / Push Down Method
Inline Variable	Extract Superclass
Change Variable Type	Split Class

Why the "well handled" types succeed. These are *local* transformations that operate within a single method or class. Rename operations require only lexical understanding. Extract Method requires identifying a coherent block of statements and their input/output variables — a pattern that LLMs learn well from training data. Simplifying conditionals (e.g., replacing nested if-else chains with guard clauses) follows well-known refactoring patterns that appear frequently in open-source code. Remove Dead Code requires recognizing unreachable statements, which LLMs can do within a visible context window.

Why the "poorly handled" types fail. These are *structural* transformations that require reasoning about relationships *between* classes, modules, or packages. Extract Class demands understanding which subset of fields and methods form a coherent abstraction — a design judgment, not a pattern match. Move Method requires knowing the dependency graph across files. Pull Up / Push Down involves inheritance hierarchies and Liskov substitution concerns. Liu et al. reported **0% recall for Extract Class** with generic prompts on both GPT-4 and Gemini [33]. Cordeiro et al. confirmed that developers outperform LLMs precisely on "complex structural" refactorings: Move Method, Extract Superclass, Pull Up Method, and Move Source Folder [35].

The pattern is clear: **LLMs excel at within-file, pattern-based transformations and fail at cross-file, design-judgment transformations.** This is the central limitation that motivates the multi-agent and tool-integrated approaches described in the next sections.



7.3 Prompt Engineering for Refactoring

7.3.1 Piao et al.: systematic instruction strategies

The most comprehensive study of prompt engineering for refactoring is Piao et al.'s evaluation of five instruction strategies across all 61 refactoring types from Fowler's catalog [34]. This study is remarkable for its breadth: rather than testing a handful of refactoring types, the authors systematically covered the entire canonical catalog.

The five strategies, along with their benchmark success rates for GPT-4o-mini and DeepSeek-V3, are shown below:

Strategy	Description	GPT-4o-mini Success	DeepSeek-V3 Success	Example Prompt Fragment
Zero-Shot	Just the refactoring name, no explanation	47.5%	91.8%	"Apply Extract Method refactoring to this code."
Two-Shot	Two worked examples before the request	57.4%	95.1%	[Example 1 before/after] + [Example 2 before/after] + "Now apply to this code:"
Step-by-Step	Human-oriented procedural instructions	83.6%	100%	"1. Identify the code block to extract. 2. Create a new method. 3. Replace the block with a call..."

Strategy	Description	GPT-4o-mini Success	DeepSeek-V3 Success	Example Prompt Fragment
Rule-based	Machine-oriented detection heuristics	80.3%	100%	“If method M has LOC > 20 and contains a block B used only once, extract B into a new method…”
Objective Learning	Only the high-level quality goal	29.5–36.1%	31.1–37.7%	“Improve the maintainability of this code.”

The **key finding** is dramatic: moving from a generic prompt to a subcategory-specific prompt boosts success from 15.6% to 86.7% (Liu et al.’s finding using GPT-4 with narrowed scope) [33]. Piao et al. confirmed a similar trajectory: zero-shot prompts achieve 47.5% on GPT-4o-mini, while step-by-step instructions reach 83.6%.

Several insights deserve special attention:

1. **Rule-based instructions outperform human-oriented step-by-step for GPT-4o-mini.** This is counterintuitive — one might expect that natural language steps would be easier for a language model. But rule-based heuristics (derived from automated refactoring detection tools like Ref-Finder) provide precise pre- and post-conditions that the LLM can follow mechanically. As Piao et al. note: “Rule-based instructions adapted from heuristics used in automated refactoring detection enable GPT-4o-mini to produce compilable, semantically preserved code more effectively than step-by-step, human-oriented instructions” [34].

2. **The Objective Learning paradox.** When given only a high-level goal (“improve maintainability”), LLMs produce the *lowest* correctness (29–36%) in terms of performing the intended refactoring type. But paradoxically, the code they produce achieves the *best* quality metrics — the lowest cyclomatic complexity and lines of code. The explanation: freed from the constraint of performing a specific transformation, the LLM applies whatever improvements it deems best, which often results in substantial restructuring that happens to improve metrics even if it does not match the intended refactoring type.

3. **DeepSeek-V3 achieves 100% on 48 of 61 Fowler types** with both Step-by-Step and Rule-based strategies. This suggests that the refactoring capability gap is closing rapidly for well-specified single-file transformations.

7.3.2 Liu et al.: prompt structure and optimization

Liu et al.’s study provides complementary evidence on prompt structure [33]. They tested GPT-4 and Gemini-1.0 Pro on 180 real-world refactorings from 20 open-source Java projects, systematically varying prompt specificity.

Their findings on prompt optimization:

- **Generic prompts** ("Identify refactoring opportunities in this code") yield only 15.6% recall for GPT-4 and 3.9% for Gemini.
- **Type-specified prompts** ("Identify Extract Method opportunities") boost recall to 52.2% for GPT-4 — a 235% improvement.
- **Subcategory + narrowed scope prompts** ("Identify Extract Method opportunities in the calculateTotal method, specifically looking for blocks that can be extracted into helper methods") achieve **86.7% recall** for GPT-4.
- **Code size effect:** A moderate negative correlation (-0.38 for GPT-4) exists between code size and refactoring quality. Practical use is confined to fragments of **<300 LOC**.

7.3.3 Practical example: the same refactoring, three prompts

To make these abstract findings concrete, consider an Extract Method refactoring applied to a Python function that calculates order totals with tax and discount logic interleaved:

Original code:

```python
def process_order(order):
    subtotal = 0
    for item in order.items:
        price = item.unit_price * item.quantity
        if item.is_taxable:
            price += price * order.tax_rate
        if item.discount_code:
            discount = lookup_discount(item.discount_code)
            price -= price * discount.percentage
        subtotal += price
    shipping = calculate_shipping(order.weight, order.destination)
    return subtotal + shipping
```

Prompt Strategy 1 — Zero-Shot:

```
Apply Extract Method refactoring to this code.
```

Typical LLM output: The model might extract the shipping calculation but leave the complex loop body untouched, or it might rename variables but not restructure the logic. Success is unpredictable.

Prompt Strategy 2 — Step-by-Step:

```
Apply Extract Method refactoring to the following code.
Steps:
1. Identify the block within the for-loop that computes the price
   for a single item (including tax and discount).
2. Extract that block into a new method called calculate_item_price
   that takes an item and the tax_rate as parameters.
3. Replace the original block with a call to calculate_item_price.
4. Ensure the new method returns the computed price.
```

Typical LLM output:

```python
def calculate_item_price(item, tax_rate):
    price = item.unit_price * item.quantity
    if item.is_taxable:
        price += price * tax_rate
    if item.discount_code:
        discount = lookup_discount(item.discount_code)
        price -= price * discount.percentage
    return price

def process_order(order):
    subtotal = sum(
        calculate_item_price(item, order.tax_rate)
        for item in order.items
    )
    shipping = calculate_shipping(order.weight, order.destination)
    return subtotal + shipping
```

Prompt Strategy 3 — Rule-Based:

```
Apply Extract Method refactoring.
Rule: If a loop body contains > 5 statements and uses local variables
that can be parameterized, extract the body into a separate method.
Pre-condition: The block must have clearly identifiable inputs
(parameters) and a single output (return value).
Post-condition: The original loop calls the new method; all variable
references are preserved; behavior is identical.
Target: The for-loop body in process_order.
```

Typical LLM output: Very similar to Strategy 2, but with more consistent handling of edge cases (e.g., correctly parameterizing `tax_rate` rather than accessing it through `order`). Rule-based prompts reduce ambiguity about *what* to extract and *how* to parameterize it.

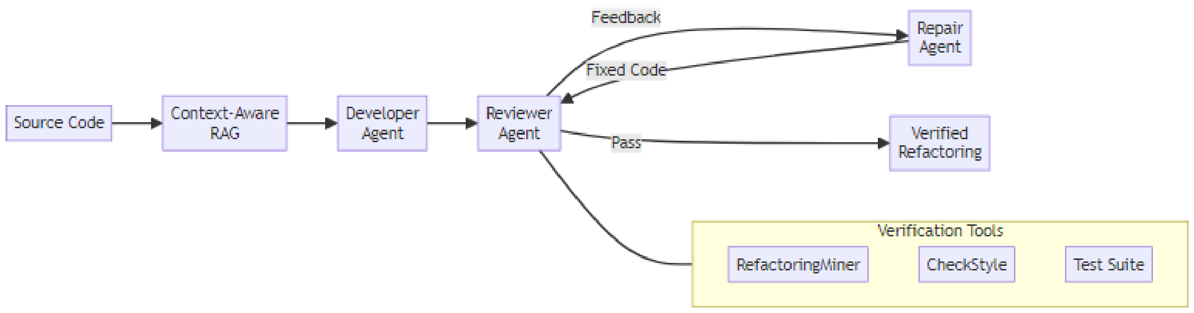
The practical takeaway: **always specify the refactoring type, the target code region, and either step-by-step instructions or formal pre/post-conditions.** Generic prompts waste tokens and produce unreliable results.

7.4 Multi-Agent Frameworks

7.4.1 MANTRA: multi-agent refactoring with RAG

The most significant advance in LLM-based refactoring to date is MANTRA (Multi-Agent eNhanced refacToring with RAG), developed by Xu et al. [9]. MANTRA addresses the central weakness of standalone LLM refactoring — the lack of project context and verification — through a coordinated multi-agent architecture.

Architecture. The following diagram illustrates MANTRA's multi-agent pipeline:



MANTRA employs three specialized agents orchestrated by LangGraph:

1. **Developer Agent** — generates the refactored code, informed by Context-Aware RAG that retrieves similar past refactorings, API documentation, and project conventions.
2. **Reviewer Agent** — validates the generated code using traditional SE tools (RefactoringMiner for verifying that the intended refactoring type occurred, CheckStyle for style compliance, compilation and test execution for functional correctness). Provides structured feedback to the Developer Agent.
3. **Repair Agent** — receives feedback from the Reviewer and iteratively fixes compilation errors, test failures, and style violations using Verbal Reinforcement Learning (Reflexion).

The RAG component is particularly important. It uses a hybrid retrieval strategy (BM25 sparse retrieval + all-MiniLM-L6-v2 dense retrieval, merged via Reciprocal Rank Fusion) to provide the Developer Agent with contextually relevant information: how similar refactorings were performed in the same project, what APIs are available, and what coding conventions are expected.

Results. MANTRA was evaluated on 703 pure structural refactorings from 10 open-source Java projects (checkstyle, pmd, commons-lang, hibernate-search, junit4, commons-io, javaparser, junit5, hibernate-orm, mockito):

Metric	MANTRA (GPT-4o-mini)	RawGPT Baseline	Improvement
Compile + Test + Verified success	82.8% (582/703)	8.7% (61/703)	9.5x
CodeBLEU similarity to human code	0.640	0.517	+23.8%
AST Diff Precision / Recall	0.781 / 0.635	0.773 / 0.574	+10.6% recall
Identical to developer refactoring	18.0%	13.1%	+37.4%

Metric	MANTRA (GPT-4o-mini)	RawGPT Baseline	Improvement
Cost per refactoring	<$0.10	—	—

The contrast between **82.8%** (MANTRA) and **8.7%** (raw GPT-4) is the single most important number in the LLM refactoring literature. It demonstrates that pipeline architecture — not model capability alone — determines success.

Ablation study. The ablation results reveal which components matter most:

Component Removed	Success Rate	Drop from Full
None (full MANTRA)	582 (82.8%)	—
Without Reviewer Agent	222	-61.9%
Without Repair Agent	287	-50.7%
Without RAG	345	-40.7%

The Reviewer Agent — which integrates traditional SE tools — contributes the largest improvement. This strongly supports a design principle for any LLM refactoring system: **external verification via traditional tools is more valuable than additional LLM reasoning.**

A user study with 37 developers found MANTRA-generated code comparable to human-written code in both readability (4.15 vs. 4.02) and reusability (4.13 vs. 3.97), with no statistically significant difference.

7.4.2 Horikawa et al.: large-scale agentic refactoring in the wild

While MANTRA studies a controlled pipeline, Horikawa et al. provide the first large-scale empirical analysis of how AI coding agents (OpenAI Codex, Claude Code, Devin, Cursor) perform refactoring in real-world open-source projects [37]. Their study analyzed 15,451 refactoring instances across 14,998 commits from the AIDev dataset — the largest corpus of AI-generated refactorings to date.

Key findings:

- **Agents are active refactorers:** 26.1% of agentic commits explicitly target refactoring.
- **Dominant types are low-level:** Change Variable Type (11.8%), Rename Parameter (10.4%), Rename Variable (8.5%). These are consistency edits, not structural improvements.
- **Agents are NOT architectural planners:** Only 3.7% of refactoring-containing changes touch more than one file. High-level structural refactorings (Extract Subclass, Split Class) are extremely rare. Agents overwhelmingly perform low-level (35.8%) and medium-level (21.2%) changes, compared to humans who perform more high-level changes (54.9% vs. 43.0%).
- **Smell reduction is negligible:** Despite 52.5% of agentic refactoring being motivated by maintainability, the median design and implementation smell delta is 0.00 — agents fix surface issues but do not address underlying design problems.

- **Medium-level refactorings produce the largest quality gains:** When agents *do* perform Extract Method or Inline Method (medium-level), the structural metrics improve substantially (class LOC median delta = -15.25, WMC median delta = -2.07).

The authors capture the current state of affairs with a vivid characterization:

> "Agentic coding tools effectively serve as incremental cleanup partners, excelling at localized refactoring and consistency improvements necessary for long-term maintainability. However, to realize the vision of agents as 'software architects,' significant advancements are needed to enable autonomous, architecturally-aware restructuring that consistently addresses higher-level design smells." [37]

This characterization — **tactical cleanup partners, not strategic architects** — precisely defines the gap that architectural tactic implementation aims to fill.

7.4.3 Cordeiro et al.: agent comparison and verification loops

Cordeiro et al. provide additional evidence on the importance of verification in agent-based refactoring [35]. Their large-scale comparison of StarCoder2-15B against human developers on 5,194 commits from 30 Java projects revealed:

- **LLMs reduce more code smell types** (44.4% smell reduction rate) compared to developers (24.3%).
- **But functional correctness is low:** Pass@1 is only 28.4% — meaning over 70% of single-attempt refactorings break tests.
- **Multiple generations help:** Pass@5 reaches 57.2%, a 28.8 percentage point improvement.
- **Chain-of-thought prompting unlocks new types:** Adding chain-of-thought reasoning introduces 7 additional refactoring types that the model cannot perform with simpler prompts, including Extract Method, Extract Class, and Parameterize Variable.

The finding that multiple generations with test validation substantially improve quality reinforces the multi-agent approach: agents with verification loops outperform single-pass generation.

7.5 Tool-Integrated Pipelines

7.5.1 Goncalves and Maia: iterative SonarQube + LLM pipeline

While multi-agent frameworks coordinate multiple LLM instances, tool-integrated pipelines combine LLMs with *existing static analysis tools* in a feedback loop. The most directly relevant example is Goncalves and Maia's iterative SonarQube + LLM pipeline [36].

Architecture. The pipeline operates as a closed feedback loop:

```
Codebase --> [SonarQube Analysis] --> Issues List
```

```
                      |
                      v
          [LLM Prompt with Issue Context]
                      |
                      v
              [Generated Fix]
                      |
                      v
          [Apply Fix to Codebase]
                      |
                      v
    [Re-analyze with SonarQube] --> Issues List --> ...
                      |
               (repeat N times)
```

Each iteration feeds the current SonarQube findings — issue type, severity, location, and description — into the LLM prompt. The LLM generates a fix, the fix is applied, and SonarQube re-analyzes the modified code. This continues until convergence or a maximum iteration count is reached.

Results. Evaluated on three open-source Java repositories (Apache Commons Lang, Apache Commons IO, Google Guava) with 8 experimental configurations varying model (GPT-4-mini, Gemini), temperature (0.1, 0.3), prompt style (zero-shot, role-based), and iteration count (2, 5):

Configuration	Issue Reduction	Best Project
Best (Gemini, temp 0.1, zero-shot, 5 iter.)	81.3%	Commons Lang
Average across all configs	>58%	—
Worst (GPT-4-mini, temp 0.3, role-based, 2 iter.)	49.5%	—

Critical nuances:

1. **Issue reduction does not equal debt reduction.** In Commons Lang, 58.8% issue reduction translated to only 42.1% technical debt reduction (ratio 0.71). Some LLM-generated fixes introduce *new*, more complex issues while resolving simpler ones. This echoes Horikawa et al.'s finding that agents create new smells while fixing others in 23% of cases [37].

2. **More iterations help, but with diminishing returns.** Five iterations consistently outperformed two iterations when other parameters were well-tuned. However, after 3–4 iterations, the remaining issues tend to be the hardest ones — BLOCKER-severity issues actually showed slight *increases* in some experiments.

3. **Temperature-prompt interaction.** Higher temperature (0.3) produces more creative but less reliable fixes. When combined with vague prompts, this combination yields the worst results. Low temperature (0.1) with specific, zero-shot prompts produces the most consistent improvements.

4. **LLMs operate as black boxes regarding architecture.** As the authors observe: "The evaluated LLMs operate as black-box systems, lacking explicit reasoning re-

> garding the software's broader architecture and context. As a result, the generated improvements are primarily focused on localized changes, potentially ignoring dependencies and interactions among components" [36].

This last point is the fundamental limitation of tool-integrated pipelines at the code level: they can iterate on *local* issues but cannot reason about *architectural* quality. SonarQube detects code smells and rule violations; it does not detect missing architectural tactics, eroded module boundaries, or coupling patterns that require structural reorganization. Extending this iterative feedback pattern to the architecture level — replacing (or augmenting) SonarQube with architecture-aware analysis — is a natural next step and a core motivation of the thesis.

7.6 Behavior Preservation

7.6.1 The fundamental challenge

Refactoring, by definition, must preserve behavior: the code after transformation must do exactly what it did before. This is easy to state but remarkably difficult to verify, especially for LLM-generated transformations where the model may subtly change semantics while appearing to preserve structure.

The challenge is compounded by the fact that different refactoring contexts demand different levels of assurance. Renaming a local variable is low-risk; extracting a method that captures mutable state is medium-risk; restructuring module boundaries is high-risk. The verification method must match the risk level.

7.6.2 Methods used across studies

The following table catalogs the behavior preservation methods used in the literature, ordered from least to most reliable:

Method	Description	Reliability	Used By
LLM self-evaluation	Ask the LLM whether behavior was preserved	Low	[32] (97.2% claimed, but potentially biased)
Compilation check	Verify that the refactored code compiles	Low	[9], [36] (necessary but not sufficient)
CodeBLEU	Semantic similarity metric comparing original and refactored code	Medium-Low	[9] (MANTRA: 0.640 similarity)
AST diff	Structural comparison of abstract syntax trees before and after	Medium	[9] (precision 0.781, recall 0.635)

Method	Description	Reliability	Used By
Unit test pass rate (Pass@K)	Run existing or generated tests; pass rate measures preservation	Medium-High	[9], [33], [35]
RefactoringMiner verification	Confirm that the intended refactoring type actually occurred	Medium-High	[9], [34] (99% precision, 94% recall)
Expert manual review	Human developers inspect the transformation for correctness	High	[33] (3 experts, Fleiss kappa = 0.82); [34] (2 researchers, Cohen kappa 0.77–0.86)
Formal verification	Mathematical proof that input/output behavior is identical	Highest	No study achieves this — future direction

7.6.3 The reliability spectrum

These methods form a spectrum from “probably correct” to “provably correct”:

LLM self-evaluation (lowest reliability): DePalma et al. reported 97.2% behavior preservation based on asking ChatGPT itself whether the refactored code preserved behavior [32]. This is a circular assessment — the model that performed the refactoring is asked to evaluate its own work. While the number is high, it cannot account for subtle semantic changes that the model itself does not recognize.

Compilation + unit tests (medium reliability): MANTRA’s 82.8% success rate is measured by the compound criterion of compilation *plus* test passage *plus* RefactoringMiner confirmation [9]. This is substantially more rigorous. However, unit tests have limited coverage — Liu et al. note that “software projects in the wild rarely have sufficient regression unit tests” [33], and semantic changes that fall outside test coverage can pass undetected. Cordeiro et al.’s finding that Pass@1 is only 28.4% but Pass@5 reaches 57.2% [35] highlights how sensitive this metric is to the number of generation attempts.

Expert manual review (high reliability): The gold standard in current practice, but inherently unscalable. Liu et al. employed three independent experts spending 42 person-days to validate 180 refactorings [33]. This level of effort is appropriate for research validation but impractical for automated pipelines.

Formal verification (highest reliability): No study in the current literature achieves formal verification of LLM-generated refactorings. This remains an open problem. The gap is significant: without formal guarantees, every LLM-generated transformation carries a residual risk of behavioral change, estimated at approximately **7% for GPT-4** [33] and **higher for smaller models** (StarCoder2-15B: 71.6% failure rate at Pass@1) [35].

The practical implication is clear: **any LLM refactoring pipeline must include automated testing as a minimum verification step, and should layer multiple verification methods** (compilation + tests + static analysis + type checking) to reduce the probability of undetected behavioral changes.

7.7 Refactoring at Scale: Industry Evidence

7.7.1 Kim, Zimmermann, and Nagappan: the Microsoft study

To ground the academic literature in industrial reality, we turn to Kim et al.'s landmark study of refactoring practices at Microsoft [38]. This study combined three complementary data sources: a survey of 328 engineers across five Microsoft products (Windows Phone, Exchange, Windows, OCS, Office), interviews with the Windows 7 refactoring team, and quantitative analysis of the Windows 7 version history.

Key findings relevant to this study guide:

1. **Top refactoring motivations:** Poor readability, code duplication, and preparation for a new feature. Note that "preparation for a new feature" is explicitly architectural — it means restructuring code to *accommodate* future changes, which is precisely what maintainability tactics aim to achieve proactively.
2. **Benefits are multi-dimensional:** The top 5% of preferentially refactored modules in Windows 7 reduced inter-module dependencies by a factor of 0.85, while non-refactored modules *increased* dependencies by a factor of 1.10. But the same refactored modules showed *increased* LOC and crosscutting changes — improvements on one dimension came at a cost on another.
3. **Metrics do not consistently capture improvements.** The same refactoring campaign that reduced coupling increased code size. A single metric (e.g., Maintainability Index) would miss the coupling improvement while penalizing the size increase. This is the strongest industrial evidence for the principle that **multiple metrics are needed, not single-metric evaluation**.
4. **Practitioners define refactoring broadly.** 46% of surveyed engineers did *not* mention behavior preservation in their definition of refactoring — they included functional changes, performance tuning, and even feature additions. This matters for LLM-based approaches: if the model is trained on commits labeled "refactoring" that actually change behavior, it may learn to conflate refactoring with modification.

7.7.2 Horikawa's AIDev dataset: the largest AI-refactoring corpus

Horikawa et al.'s AIDev dataset provides the largest empirical window into AI-generated refactoring [37]. Across 15,451 refactoring instances:

- **Smell Reduction Rate varies dramatically by smell type:** Long Method achieves 73% reduction (agents are good at breaking up large methods), but Feature Envy shows only 12% reduction (agents cannot reliably move methods to the class they "envy").
- **Agents create new smells while fixing others in 23% of cases.** This is not merely a failure to improve — it is active degradation. A fix that reduces cyclomatic complexity by extracting a method may simultaneously introduce a God Class smell by adding the method to an already-bloated utility class.

- **86.9% of agentic pull requests are merged,** indicating high acceptance despite the limited quality impact. This suggests that developers value the time savings of agent-generated cleanups even when the architectural impact is marginal.

The contrast between high merge rates and negligible smell reduction raises an important question: are developers accepting low-impact changes because they lack the time or tools to evaluate architectural quality? This is exactly the situation that automated architectural tactic implementation could address.

7.8 The Code-Level vs. Architecture-Level Gap

7.8.1 All current LLM refactoring operates at method/class level

After surveying the entire landscape of LLM-based refactoring — from standalone models through prompt engineering to multi-agent frameworks and tool-integrated pipelines — one conclusion is inescapable: **no existing work uses LLMs to implement architectural tactics.** Every study reviewed in this chapter operates at the level of individual methods, single classes, or at most pairs of classes. The "architecture" in these studies refers to the architecture of the *pipeline* (multi-agent, iterative feedback), not to the architecture of the *target software*.

Martinez et al.'s systematic literature review confirms this gap explicitly: the taxonomy of LLM-refactoring research covers code smell removal, method-level restructuring, and test generation — but not architectural restructuring, tactic implementation, or system-level quality improvement [31].

7.8.2 The gap in detail

The following table crystallizes the difference between what current tools can do and what the thesis aims to achieve:

Aspect	Code-Level (Current)	Architecture-Level (Thesis Gap)
Scope	Single method or class	Multiple modules and packages
Context needed	Local (one file, <300 LOC)	System-wide (project structure, dependency graph)
Typical examples	Extract Method, Rename Variable, Inline Method	Split Module, Use Intermediary, Restrict Dependencies
Design judgment	Minimal (pattern matching)	Substantial (quality attribute trade-offs)
Verification	Unit tests sufficient	Integration tests + architecture conformance checks
Best success rate	82.8% (MANTRA) [9]	Unknown — no empirical data exists
Available tools	RefactoringMiner, CheckStyle, PMD	None purpose-built for architecture-level LLM refactoring

Aspect	Code-Level (Current)	Architecture-Level (Thesis Gap)
Files touched	1 (96.3% of agentic refactorings) [37]	5–15+ (typical tactic implementation)
Behavior preservation	Unit test pass rate	Unit + integration test pass rate + architectural constraint validation
Risk level	Low to medium	High (structural changes affect entire system)

7.8.3 Why the gap exists

Three factors explain why LLM refactoring has remained at the code level:

1. **Context window limitations.** Architecture-level transformations require understanding the entire project structure — its module boundaries, dependency graph, configuration files, and test infrastructure. Even with 128K-token context windows, representing a full project is challenging. Current approaches work within single files precisely because they fit within the context window.

2. **Lack of architecture-aware tools.** RefactoringMiner verifies that an Extract Method occurred; no equivalent tool verifies that a "Use Intermediary" tactic was correctly implemented. CheckStyle enforces coding conventions; no equivalent enforces architectural constraints. Without verification tools, multi-agent feedback loops — which we have seen are critical to success — cannot operate at the architecture level.

3. **Training data bias.** LLMs are trained on code commits, and the vast majority of refactoring commits are code-level changes. Kim et al. found that even at Microsoft, architecture-level refactoring is rare and poorly documented [38]. Horikawa et al. confirmed that only 3.7% of agentic refactorings touch more than one file [37]. The training distribution simply does not contain enough architecture-level transformations for models to learn the patterns.

7.8.4 The thesis opportunity

This gap is not merely an observation — it is the research opportunity that motivates the entire thesis. The evidence in this chapter establishes several foundational facts:

- LLMs *can* transform code reliably when provided with specific instructions and verification feedback (Section 7.3, 7.4).
- Multi-agent pipelines with tool integration achieve dramatically higher success than standalone models (Section 7.4, 7.5).
- The iterative feedback pattern (static analysis -> LLM fix -> re-analysis) works at the code level and can plausibly be extended to the architecture level (Section 7.5).
- Current agents are "tactical cleanup partners, not strategic architects" [37] — but nothing in principle prevents them from becoming architectural partners if given the right context, instructions, and verification tools.

Chapter 9 will describe the specific research design that aims to bridge this gap: a pipeline that provides LLMs with architectural tactic specifications (the "what"), project-level context via static analysis (the "where"), and architecture-conformance verification (the "did it work") — extending the patterns established by MANTRA and the SonarQube pipeline from code-level to architecture-level transformations.

7.9 Review Questions

1. **Comprehension.** Why does MANTRA achieve 82.8% success while a raw GPT-4 prompt achieves only 8.7% on the same refactoring tasks? Which component of MANTRA contributes the most, and why?
2. **Analysis.** DePalma et al. report 97.2% behavior preservation for ChatGPT refactoring, while Cordeiro et al. report only 28.4% Pass@1 for StarCoder2. Are these numbers contradictory? Explain what methodological differences account for the discrepancy.
3. **Evaluation.** Piao et al. found that Objective Learning prompts produce the best quality metrics but the lowest correctness. Under what circumstances would you prefer Objective Learning over Step-by-Step prompting in a real refactoring pipeline?
4. **Synthesis.** Design a behavior preservation strategy for an LLM refactoring pipeline that balances thoroughness with scalability. Which methods from Table 7.6 would you combine, and in what order?
5. **Application.** Given the evidence that LLMs struggle with Extract Class (0% recall with generic prompts), propose a multi-step approach — combining prompt engineering, RAG, and tool verification — that might improve success for this specific refactoring type.
6. **Critical thinking.** Horikawa et al. found that 86.9% of agentic refactoring PRs are merged despite negligible smell reduction. What does this tell us about how developers evaluate refactoring contributions? What risks does high merge acceptance of low-impact changes pose for long-term maintainability?
7. **Gap identification.** Create a table listing three specific architectural tactics (from Chapter 4) and explain, for each, why current LLM refactoring tools cannot implement them. What would need to change in the pipeline architecture to make each one feasible?

Chapter 8

Challenges and Limitations

Learning objectives. After reading this chapter you should be able to (1) identify and explain the five core technical challenges that limit LLM-based architectural transformation, (2) describe why static analysis tool disagreement undermines confidence in quality measurements, (3) articulate the detection-remediation gap and its implications for automated architecture improvement, and (4) evaluate the practical obstacles — from developer trust to behavior preservation — that any real-world deployment must address.

The preceding chapters have established that architectural tactics are powerful instruments for improving maintainability, that architecture erosion silently degrades software quality, and that LLMs show genuine promise for automated code refactoring. This chapter confronts the other side of the story: the obstacles that currently prevent us from simply pointing an LLM at a codebase and asking it to "implement the right architectural tactics." These challenges are not minor inconveniences; they are fundamental barriers that shape the design of any viable automated architecture-improvement pipeline.

We organize the discussion into three categories: technical challenges inherent to LLM-based architecture transformation (Section 8.1), challenges in measuring whether the transformation actually helped (Section 8.2), and practical challenges that arise when deploying such systems in real development workflows (Section 8.3).

8.1 Technical Challenges for LLM-Based Architecture Transformation

Architectural tactics are, by definition, cross-cutting design decisions that affect multiple modules and quality attributes simultaneously [1]. This makes them fundamentally different from the local, syntactic code edits at which LLMs currently excel. Below we examine the five core technical challenges that any LLM-based tactic implementation system must overcome, providing for each one a description, evidence from the literature, an assessment of severity, and potential mitigation strategies.

8.1.1 Context Blindness

Description. LLMs process text in a bounded context window. When applied to code, they typically see one file at a time — or, at best, a small collection of files that fits within the token budget. They have no inherent understanding of the system-wide architectural context: the dependency graph, the module boundary map, the import structure, the runtime call chains, or the design rationale that explains why modules are partitioned the way they are. For architecture-level changes — which by definition span module boundaries — this creates a fundamental mismatch between the scope of the problem and the scope of the LLM's awareness.

Evidence. The MANTRA framework addresses this gap by introducing a Context-Aware Retrieval-Augmented Generation (RAG) component that retrieves relevant code from the broader repository before the LLM generates a transformation. The ablation study quantifies the impact: removing RAG reduces the success rate from 82.8% to a level that represents a 40.7% decrease, confirming that external context injection is essential rather than optional [9]. Horikawa et al.'s large-scale empirical study of AI coding agents (OpenAI Codex, Claude Code, Devin, Cursor) provides complementary evidence: agents touch only a single file in the vast majority of their refactoring actions, and their refactoring repertoire is overwhelmingly dominated by low-level, within-file edits (renaming variables, changing types) rather than cross-file structural changes [37]. When agents do attempt structural decomposition (e.g., Extract Subclass, Split Class), the quality gains are large — but such actions are rare precisely because the agents lack the architectural awareness to initiate them.

Severity. High. Architectural tactics inherently require multi-file, multi-module understanding. A tactic like "Use an Intermediary" requires knowing which modules currently depend on each other, creating a new intermediary module, and rewriting import statements and call sites across potentially dozens of files. Without system-wide context, the LLM cannot even identify where the tactic should be applied, let alone implement it correctly.

Mitigation. Several strategies can partially address context blindness:

- **Retrieval-Augmented Generation (RAG):** Retrieve relevant files, dependency information, and architectural documentation before prompting the LLM, as MANTRA demonstrates [9].
- **Context injection:** Feed the LLM a structured representation of the system architecture (dependency graph, module map, quality profile) as part of the prompt, compressing thousands of lines of code into a tractable summary.
- **Multi-agent architectures:** Assign a dedicated "context-gathering agent" that analyzes the codebase and produces an architectural summary, which is then consumed by a "transformation agent" that generates code changes.

8.1.2 Hallucinations

Description. LLMs are generative models that produce statistically plausible output — but "plausible" does not mean "correct." In code generation, hallucinations manifest as references to nonexistent variables, methods, APIs, classes, or modules. The LLM may generate a call to `subject.hasRole("admin")` when no such method exists in the codebase's security framework, or import a module that was never defined. These hallucinations are particularly dangerous for architecture-level changes because they often involve framework-specific APIs (dependency injection containers, ORM configurations, security frameworks) where the correct API surface is large and version-dependent.

Evidence. The IPSynth study provides the most striking evidence. When ChatGPT was tasked with implementing JAAS authentication tactics in 20 Java programs, it produced syntactically correct (compilable) code 95% of the time — but only **5% of implementations were semantically correct** [5]. The dominant failure mode was generating code for *part* of the tactic while leaving the rest to the programmer, and hallucinating nonexistent API calls (e.g., `subject.hasRole(...)` instead of the correct JAAS `Subject.doAs(...)` pattern). Piao et al. report a similar pattern: most compilation errors in their real-scenario experiments were caused by LLMs referencing undeclared variables or methods — hallucinations from insufficient context [34].

Severity. Critical. Unlike a naming mistake that produces a compiler error, a semantic hallucination can produce code that compiles and runs but implements the wrong behavior. In the security tactic domain, this means a system that *appears* to authenticate users but actually does not — a silent, dangerous failure.

Mitigation.

- **Compilation and type-checking feedback loops:** Run the LLM-generated code through the compiler immediately and feed errors back to the LLM for correction, as MANTRA's Repair Agent does [9].
- **Static analysis verification:** Use tools like RefactoringMiner, CheckStyle, or Radon to verify that the generated code meets structural expectations.
- **Test execution:** Run the project's existing test suite after each transformation to catch behavioral regressions.
- **Formal specification matching:** For well-defined tactics, compare the generated code against a formal specification (as IPSynth does with its FSpec model) to verify correct API usage [5].

8.1.3 Token and Length Limits

Description. LLM performance degrades as input length increases. While modern models support context windows of 100K+ tokens, the quality of code understanding and generation measurably declines for longer inputs. Architecture-level changes often require understanding thousands of lines of code spread across many files — far beyond the effective attention span of current models.

Evidence. Liu et al. report a moderate negative correlation between code size and LLM refactoring success: correlation coefficients of -0.38 for GPT-4 and -0.28 for Gemini, with practical effectiveness confined to code segments under approximately 300 lines of code [33]. Beyond this threshold, LLMs increasingly fail to identify refactoring opportunities and produce lower-quality solutions. This is particularly problematic for architectural tactics, which may require coordinated understanding of files totaling thousands of lines.

Severity. High. A single Python module implementing a complex service can easily exceed 300 LOC, and understanding the module's role in the system architecture requires examining its callers, callees, and configuration files — potentially 10,000+ LOC in aggregate.

Mitigation.

- **Chunking strategies:** Break the system into logical units (modules, packages) and process each unit separately, maintaining a shared context summary across chunks.
- **Hierarchical prompting:** First prompt the LLM to produce an architectural plan (which files need to change, what changes each file needs), then execute each file-level change in a separate prompt with focused context.
- **Context summarization:** Summarize large files or modules into compact representations (function signatures, class hierarchies, dependency lists) that convey architectural structure without consuming the full token budget.

8.1.4 The Complexity Gap

Description. LLMs perform well on local, syntactic transformations — Rename Variable, Extract Method, Inline Variable — but struggle with cross-cutting, semantic transformations that require understanding design intent and coordinating changes across multiple code locations. This creates a "complexity gap" between the kinds of refactorings LLMs can reliably perform and the kinds of transformations that architectural tactics require.

Evidence. Piao et al. evaluated LLMs across all 61 refactoring types in Fowler's catalog and found that success rates are highly type-dependent. On simple refactoring types (variable-level operations), models achieved high success rates, while complex refactoring types requiring cross-cutting changes showed significantly lower performance. In benchmark scenarios, DeepSeek-V3 achieved 100% on some simple types but real-scenario compilation rates ranged only 38–51% across all strategies [34]. Horikawa et al. confirm this at scale: agents perform 35.8% low-level refactorings versus only 21.2% medium-level and 43.0% high-level (but the high-level category is dominated by simple type changes, not architectural restructuring). The study concludes that agents are "incremental cleanup partners" rather than "software architects" [37]. Liu et al. found that both GPT-4 and Gemini achieved 0% recall for Extract Class identification with generic prompts — the most architecturally relevant refactoring type [33].

Severity. High. Architectural tactics like Split Module, Use an Intermediary, and Abstract Common Services are inherently complex, multi-location transformations. If LLMs cannot reliably perform Extract Class (a single-step structural refactoring), they are unlikely to succeed at multi-step tactic implementations without significant scaffolding.

Mitigation.

- **Decomposition:** Break complex architectural transformations into sequences of simpler refactorings that LLMs can handle individually. For example, "Use an Intermediary" could be decomposed into: (1) create a new class, (2) move relevant methods into it, (3) update import statements, (4) replace direct calls with calls through the intermediary.
- **Rule-based instruction:** Piao et al. found that formulating transformations as machine-oriented rules (pre/post conditions) rather than natural language descriptions significantly improves LLM performance [34].
- **Hybrid approaches:** Use the LLM for the generative parts of the transformation (writing new code) and traditional tools for the mechanical parts (updating imports, renaming references).

8.1.5 Non-Determinism

Description. LLMs are stochastic models. The same prompt, applied to the same code, can produce different outputs across runs. This non-determinism is problematic for automated pipelines that need consistent, reproducible results — especially when the pipeline includes multiple stages where each stage's output feeds into the next.

Evidence. Multiple studies document this challenge. DePalma et al. note that ChatGPT's unpredictability makes it "difficult to fully assess capabilities," as the same prompt yields different results across runs [32]. MANTRA addresses this by setting temperature to 0 to reduce variability, though this does not eliminate it entirely [9]. Piao et al. run each experiment 5 times per configuration to account for stochastic variation [34]. The practical implication is that an automated pipeline cannot guarantee that a transformation that succeeded in testing will produce the same result in production.

Severity. Medium. While temperature control and majority voting can reduce variance, they cannot eliminate it. For safety-critical systems or regulated environments where reproducibility is mandatory, non-determinism remains a fundamental concern.

Mitigation.

- **Temperature = 0:** Set the LLM's sampling temperature to zero (or near-zero) to minimize randomness, as MANTRA does [9].
- **Majority voting:** Run the same prompt multiple times and select the most common output, or use an ensemble of models.
- **Deterministic verification:** Accept non-deterministic generation but require deterministic verification (compilation, testing, static analysis) before accepting any output.

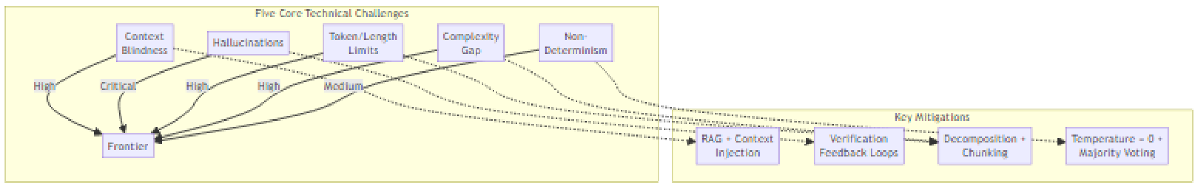
8.1.6 Summary of Technical Challenges

The following table consolidates the five technical challenges:

Challenge	Severity	Key Evidence	Current Best Mitigation	Open Problem?
Context Blindness	High	MANTRA needs RAG for 40.7% of success [9]; agents touch 1 file in vast majority of edits [37]	RAG, context injection, multi-agent	Yes — no system achieves full architectural awareness
Hallucinations	Critical	IPSynth: ChatGPT 5% semantic correctness on tactic synthesis [5]	Compilation checks, type checking, test execution, formal spec matching	Yes — semantic hallucinations evade compilers

Challenge	Severity	Key Evidence	Current Best Mitigation	Open Problem?
Token/Length Limits	High	Performance degrades beyond ~300 LOC [33]	Chunking, hierarchical prompting, context summarization	Partially — improving with larger context windows
Complexity Gap	High	0% Extract Class recall with generic prompts [33]; agents are "cleanup partners" not "architects" [37]	Decomposition into simpler steps, rule-based instructions	Yes — no LLM handles multi-step architectural refactoring
Non-Determinism	Medium	Same prompt yields different outputs [32], [34]	Temperature = 0, majority voting, deterministic verification	Partially — mitigated but not eliminated

Two challenges — hallucinations and the complexity gap — are currently open problems without satisfactory solutions. These define the frontier that any LLM-based architectural tactic implementation system must push against.



8.2 Measurement Challenges

Even if an LLM successfully implements an architectural tactic, how do we know whether the transformation actually improved the system? Measuring software quality is harder than it appears, and several methodological challenges undermine confidence in before/after comparisons.

8.2.1 Tool Disagreement

Lenarduzzi et al. conducted the largest empirical comparison of six widely used static analysis tools (SonarQube, Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD) applied to 47 Java projects [28]. The results are sobering:

- **Overall inter-tool detection agreement: less than 0.4%.** The best pairwise agreement was 9.378% (FindBugs–PMD at class level); the worst was 0.144% (CheckStyle–PMD).
- **Precision varies dramatically:** SonarQube had the broadest rule coverage but the lowest precision (18%); CheckStyle had the highest precision (86%) but detected

mostly syntactic/formatting issues.

- **Only 6 out of 66 rule pairs** showed 100% agreement on detecting the same issue — all between CheckStyle and PMD.

The implication is stark: **any metric improvement measured by a single tool may not be confirmed by another tool.** If a study reports that LLM-applied tactics reduced SonarQube violations by 30%, a reader cannot assume that an equivalent improvement would appear in PMD, FindBugs, or any other tool. This is not a minor calibration issue — it is a fundamental disagreement about what constitutes a quality problem.

For researchers and practitioners designing evaluation frameworks, this finding demands a multi-tool validation strategy. Relying on a single tool — even a respected one like SonarQube — introduces both false positive risk (the tool flags issues that are not real problems) and blind spot risk (the tool misses issues that other tools catch).

8.2.2 No Tactic-Specific Metrics

There is currently no standardized, widely accepted way to measure whether an architectural tactic was *correctly implemented*. We have metrics for code complexity (cyclomatic complexity), coupling (fan-in/fan-out, CBO), cohesion (LCOM), and aggregate maintainability (Maintainability Index). But none of these directly answers the question: "Was the Use an Intermediary tactic correctly applied?"

Architecture conformance checking tools (e.g., Reflexion Modelling, Lattix, Structure101) can verify that module dependencies match an intended architecture [4], but these require a manually specified intended architecture as input — which may not exist, may be outdated, or may be ambiguous. ArchTacRV [39] proposes ML-based tactic detection combined with runtime behavioral verification against RBML specifications, but it focuses on *detecting existing* tactics rather than *verifying newly implemented* ones, and requires manually authored RBML specifications for each tactic type.

The absence of tactic-specific metrics means that researchers must rely on proxy indicators — improvements in coupling, cohesion, or complexity that are *consistent with* the expected effect of a tactic, but do not conclusively prove that the tactic was correctly implemented.

8.2.3 Dataset Limitations

The benchmarks available for evaluating LLM-based architectural transformations suffer from three systematic limitations:

1. **Java domination.** Nearly all existing datasets and benchmarks are Java-only. IPSynth uses Java/JAAS [5]. MANTRA evaluates on 10 Java projects [9]. Piao et al. translate Fowler's JavaScript examples to Java for evaluation [34]. Liu et al. restrict their study to Java because refactoring detection tools do not support other languages [33]. Horikawa et al. analyze only Java files from the AIDev dataset [37]. This means we have virtually no empirical evidence on how LLM-based transformations perform on Python, JavaScript, Go, Rust, or any other language.

2. **Small scale.** IPSynth's benchmark consists of only 20 tasks [5]. Liu et al.'s carefully curated dataset has 180 refactoring instances — impressive in rigor but small in

statistical power [33]. Even MANTRA's 703-instance dataset, the largest, covers only 6 refactoring types across 10 projects [9].

3. **No architecture-level transformation benchmarks.** Every existing benchmark operates at the method or class level. There are no publicly available datasets of systems before and after architectural tactic implementation, with ground truth labels indicating which tactic was applied, where, and what the expected quality improvement should be.

8.2.4 The Metric Paradox

Kim et al.'s field study at Microsoft provides a cautionary finding: **the same refactoring can improve one metric while worsening another** [38]. In the Windows 7 codebase, modules that underwent intensive refactoring reduced inter-module dependencies by a factor of 0.85 but *increased* lines of code and cross-cutting changes. This means that a simple "did the metric go up or down?" evaluation is insufficient.

The metric paradox is especially acute for architectural tactics, which often involve structural trade-offs. Introducing an intermediary (a new class that mediates between modules) reduces coupling but increases the total number of classes, increases LOC, and may introduce a new point of failure. A naive metric evaluation would see the LOC increase and conclude the transformation was harmful, while a more nuanced analysis would recognize the coupling reduction as the intended benefit.

This demands multi-dimensional evaluation: any assessment of architectural tactic impact must track multiple complementary metrics (coupling, cohesion, complexity, LOC, dependency counts) and interpret them in the context of the specific tactic's intended effect.

8.3 Practical Challenges

Technical feasibility and measurement validity are necessary but not sufficient. Even a perfectly functioning LLM-based tactic implementation system faces practical challenges when deployed in real development workflows.

8.3.1 The Detection-Remediation Gap

Rosik et al.'s 2-year longitudinal case study at IBM demonstrated a troubling pattern: **none of the 9 identified architectural violations were fixed by the development team**, despite being explicitly detected and reported [8]. Developers were aware of the drift. They could see the violations in the Reflexion Model output. But they chose not to fix them because:

- **Risk of ripple effects:** Fixing one violation might break other parts of the system.
- **Time pressure:** The effort required to fix the violation exceeded the perceived benefit.
- **Legacy code entanglement:** Violations were intertwined with code inherited from previous versions.

This finding reveals a fundamental gap between *detection* and *remediation*. The software architecture community has invested heavily in detection (conformance checking tools, smell detectors, metric dashboards), but detection alone does not produce improvement. As Rosik's team observed:

> "Maybe trying to fix these 'minor' issues would've possibly caused larger issues to appear and so made them not worth exploring..." [8]

Automated remediation — using LLMs to not just identify but also *implement* the fix — could potentially break this cycle by reducing the cost and risk of addressing detected violations. But this requires the LLM's output to be trustworthy enough that developers are willing to accept it, which leads to the next challenge.

8.3.2 Developer Trust and Adoption

Developers are, rightly, skeptical of automated architectural changes. Architecture-level modifications are among the highest-risk transformations in software engineering: they touch many files, affect many stakeholders, and can introduce subtle behavioral changes that are difficult to detect.

For automated architecture transformation to gain adoption, the system must provide:

- **Explainability:** Not just "here is the new code," but "here is *why* this change was made, *which tactic* it implements, and *what quality improvement* it targets." The LLM must generate documentation alongside code.
- **Diff-based review:** Changes should be presented as reviewable diffs, not wholesale file replacements, so developers can inspect exactly what changed.
- **Incremental application:** Tactics should be applied in small, individually reviewable steps — not as a single monolithic transformation that changes 50 files at once.
- **Rollback capability:** Every transformation must be reversible. If the tactic implementation introduces unexpected problems, the developer must be able to return to the previous state instantly.
- **Confidence scores:** The system should report its confidence in each transformation, flagging uncertain changes for human review while applying confident changes automatically.

DePalma et al.'s user study found that while 85.8% of participants rated ChatGPT's refactoring quality at 5 or 6 out of 7, **92.9% said additional refactoring was still needed** [32]. This suggests that developers see LLMs as useful starting points but not as replacements for human judgment — a "suggestive auxiliary tool" rather than an autonomous architect.

8.3.3 Behavior Preservation vs. Architecture Improvement

The standard assumption in refactoring research is that transformations must be strictly behavior-preserving: the system's external behavior must be identical before and after the change. This assumption works well for method-level refactorings (Rename Variable, Extract Method) where the external API is unchanged.

But architecture-level transformations may *legitimately require behavioral changes.* Consider:

- **Introducing an intermediary:** The module interfaces change — callers now go through the intermediary instead of calling the original module directly. The functional outcome is the same, but the calling API is different.
- **Splitting a module:** A single module becomes two modules with a new inter-module interface. Existing clients must be updated to use the correct half.
- **Abstracting common services:** Concrete implementations are replaced by abstract interfaces. Client code must be updated to depend on the abstraction rather than the concrete class.

In each case, the system's *functionality* is preserved (the same inputs produce the same outputs), but the *interfaces* change. Current LLM evaluation frameworks — which compare before and after behavior through test execution — may incorrectly flag these legitimate interface changes as failures.

What is needed is a notion of "controlled behavioral change" rather than strict preservation: the system's functional behavior is preserved, but its structural interfaces are allowed to evolve in well-defined ways. Formalizing this notion and building verification tools around it is an open research challenge.

Liu et al. quantify the risk: 7.4% of GPT-4's refactoring solutions introduced unsafe changes (semantic bugs or syntax errors), with semantic changes being more dangerous because they evade compiler checks [33]. For architecture-level changes that intentionally alter interfaces, distinguishing "intentional interface change" from "accidental semantic bug" requires understanding the designer's intent — precisely the kind of reasoning that LLMs currently struggle to articulate.

8.4 Chapter Summary

The challenges described in this chapter are not reasons to abandon the pursuit of LLM-based architectural tactic implementation. Rather, they define the engineering requirements for a viable system. A successful pipeline must:

1. **Inject architectural context** into the LLM's reasoning (addressing context blindness).
2. **Verify every generated transformation** through compilation, testing, and static analysis (addressing hallucinations).
3. **Decompose complex tactics** into sequences of simpler, manageable transformations (addressing the complexity gap and token limits).
4. **Use multiple complementary metrics** interpreted in the context of each tactic's intended effect (addressing measurement challenges).
5. **Present changes as reviewable, incremental, and reversible diffs** with explanatory documentation (addressing trust and adoption).

The field is not at a dead end. MANTRA has shown that multi-agent pipelines with verification feedback loops can achieve 82.8% success at method-level refactoring [9]. Goncalves et al. have demonstrated that iterative SonarQube-LLM feedback loops reduce code issues by over 58% [36]. The challenge now is to extend these approaches from code-level to architecture-level transformations — which is precisely the research gap explored in the next chapter.

Review questions.

1. Why is “context blindness” particularly problematic for architectural tactics, as opposed to method-level refactorings? What evidence supports this claim?
2. Explain the difference between syntactic correctness and semantic correctness, using the IPSynth ChatGPT evaluation as an example. Why is this distinction critical for tactic implementation?
3. If a study reports that an LLM-applied tactic reduced SonarQube violations by 30%, what caveats should a careful reader consider, given the tool disagreement findings of Lenarduzzi et al.?
4. Rosik et al. found that 0 out of 9 detected violations were fixed by developers. How could automated remediation change this outcome? What risks does it introduce?
5. Why might strict behavior preservation be an inappropriate evaluation criterion for architecture-level transformations? Give a concrete example.

Chapter 9

Research Gaps and Future Directions

Learning objectives. After reading this chapter you should be able to (1) describe the “transformation gap” between tactic detection and LLM-based code refactoring, (2) enumerate the specific research gaps that remain open in the literature, (3) evaluate the available datasets and benchmarks for future work, (4) outline a concrete five-point future research agenda, and (5) synthesize the journey of this entire study guide into a coherent understanding of the field’s trajectory.

The previous chapter catalogued the challenges that confront LLM-based architecture transformation. This chapter asks a different question: *What has not yet been attempted?* By mapping the boundaries of existing work, we can identify the gaps that represent the highest-impact opportunities for advancing both research and practice. We begin with the core contribution gap that motivates the thesis, then enumerate specific open questions, survey available datasets, and conclude with a forward-looking research agenda.

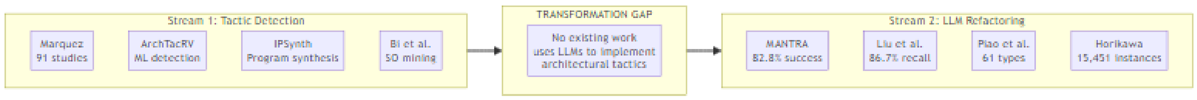
9.1 The Transformation Gap

The most significant finding from our survey of the literature is that two mature, active research streams exist in almost complete isolation from each other.

Stream 1: Tactic Detection. A substantial body of work focuses on identifying which architectural tactics exist (or should exist) in a codebase. Marquez et al.’s systematic mapping study of 91 primary studies catalogues the full landscape of architectural tactics research, documenting 12 quality attributes, 10 tactic taxonomies, and the techniques used for tactic identification (manual mapping, code analysis, text analysis, ML-based classification) [20]. Ge et al.’s ArchTacRV tool uses machine learning to detect behavioral methods of tactic structures in code and then verifies their runtime consistency against RBML specifications [39]. Bi et al. mine architecture tactic and quality attribute knowledge from 4,195 Stack Overflow posts, revealing relationships between 21 tactics and 8 quality attributes [22]. Shokri et al.’s IPSynth goes furthest, actually *implementing* tactics using program synthesis — but with a formal specification model (FSpec) and SMT solver, not an LLM [5].

Stream 2: LLM Code Refactoring. A parallel body of work demonstrates that LLMs can transform code to improve quality. MANTRA achieves 82.8% success at method-level refactoring using a multi-agent LLM pipeline [9]. Liu et al. show that LLMs with optimized prompts can identify 86.7% of refactoring opportunities and produce solutions rated comparable to human experts 63.6% of the time [33]. Piao et al. evaluate LLMs across all 61 of Fowler's refactoring types, finding that rule-based instructions outperform descriptive ones [34]. Horikawa et al. analyze 15,451 real-world refactoring instances by AI coding agents, establishing that agents are active refactoring participants — but at the code level, not the architecture level [37].

The gap is between them. No existing work uses LLMs to implement architectural tactics. The detection stream knows *what* tactics to apply but has no LLM-based mechanism to *apply* them. The refactoring stream can *transform code* but lacks the architectural awareness to select and implement *tactics*. The following diagram illustrates this disconnect and the contribution that bridging it would represent:


Stream 1: Tactic Detection
Marquez 91 studies
ArchTacRV ML detection
IPSynth Program synthesis
Bi et al. SO mining
TRANSFORMATION GAP
No existing work uses LLMs to implement architectural tactics
Stream 2: LLM Refactoring
MANTRA 82.8% success
Liu et al. 86.7% recall
Piao et al. 61 types
Horikawa 15,451 instances

The evidence for this gap is unambiguous:

- **IPSynth** achieves 85% success on tactic implementation — but uses program synthesis (FSpec + SMT solver), not LLMs. When the same tasks were given to ChatGPT, only 5% were semantically correct [5].
- **MANTRA** achieves 82.8% success on code refactoring — but operates at the method level (Extract Method, Move Method, Inline Method), not the architecture level [9].
- **Horikawa's agents** are empirically characterized as "tactical cleanup partners" that excel at localized consistency improvements but are not "software architects" capable of higher-level structural design [37].
- **Marquez's mapping** of 91 studies found that **71% do not even describe how tactics were identified**, let alone how they might be implemented automatically [20].
- **Martinez et al.'s SLR** of LLM-based refactoring research explicitly identifies the absence of architecture-level LLM refactoring as a key gap in the field [31].

Bridging this gap — building systems that use LLMs to implement architectural tactics in real codebases, verified by multi-tool static analysis — is the central research contribution that remains to be made.

9.2 Specific Research Gaps

Beyond the overarching transformation gap, the literature reveals seven specific research questions that remain unanswered. Each is grounded in empirical evidence.

#	Gap	Evidence	Research Question
1	**70% of studies lack tactic identification methods.** Most architectural tactic research does not describe how tactics were identified, making replication impossible and automation difficult.	Marquez et al. found that 65 of 91 primary studies (71%) do not describe their identification technique [20].	How can tactic identification be systematically automated using LLMs or ML, producing reproducible results?

#	Gap	Evidence	Research Question
2	**Design rationale does not trace to code.** Architects express intent as quality attribute requirements and tactic selections, but there is no automated mechanism to translate this intent into concrete code changes.	Multiple studies document the gap between design decisions and implementation [4], [7]. Rosik et al. show that even detected violations are not remediated [8].	How can LLMs bridge the gap between architectural intent (tactic specification) and code-level implementation (multi-file transformation)?

#	Gap	Evidence	Research Question
3	**No cost-benefit quantification for tactic implementation.** While tactics are qualitatively associated with quality attributes, the measurable impact of implementing a specific tactic in a specific codebase has never been systematically quantified.	Bogner et al. map 15 modifiability tactics to SOA/Microservices patterns but acknowledge that the mapping is qualitative, not quantitative [18]. Kim et al. show that refactoring effects are multi-dimensional and metric-dependent [38].	What is the measurable, multi-metric impact of implementing each modifiability tactic, and how does it vary across codebases?

#	Gap	Evidence	Research Question
4	**LLM + static analysis integration is ad hoc.** Several studies combine LLMs with static analysis tools, but each builds its own bespoke pipeline with different tools, configurations, and feedback mechanisms.	Goncalves et al. build a SonarQube-LLM loop but acknowledge that SonarQube as the sole metric misses architectural issues [36]. MANTRA uses RefactoringMiner + CheckStyle. No standardized integration framework exists.	How can we create a systematic, reusable feedback loop between LLMs and multiple static analysis tools for architecture-level transformations?

#	Gap	Evidence	Research Question
5	**No Python-specific architectural tactic benchmarks.** Nearly all existing datasets and benchmarks are Java-only, despite Python being one of the most widely used languages for backend and data-intensive systems.	IPSynth: Java/JAAS [5]. MANTRA: Java [9]. Liu et al.: Java [33]. Piao et al.: Java [34]. Horikawa et al.: Java [37].	Can we create reusable, publicly available benchmarks for architectural tactic implementation in Python and other underrepresented languages?

#	Gap	Evidence	Research Question
6	**Agents do not plan architecturally.** Current AI coding agents (Codex, Claude Code, Devin, Cursor) perform refactoring reactively — responding to individual code issues — rather than proactively planning architecture-level improvements.	Horikawa et al. find agents overwhelmingly perform low-level edits (renaming, type changes) and recommend equipping agents with design-smell detection tools to enable higher-level reasoning [37].	How can we elevate AI coding agents from tactical cleanup partners to strategic architectural planners?

#	Gap	Evidence	Research Question
7	**Behavior preservation is not formally verified.** LLM-generated transformations are tested empirically (via test suites) but never formally verified. Architecture-level changes that legitimately alter interfaces are particularly problematic.	Liu et al. report 7.4% unsafe transformation rate [33]. IPSynth uses a "correct by construction" approach with SMT solving but only for loop-free, single-framework code [5].	Can lightweight formal methods (AST-based structural verification, refinement checking) be integrated with LLM output to provide stronger guarantees?

9.3 Available Datasets and Benchmarks

For researchers entering this field, the following table summarizes the currently available datasets and benchmarks relevant to LLM-based architectural transformation. Note the absence of any architecture-level transformation dataset — which is itself a gap identified in Section 9.2.

Dataset	Size	Scope	Language	Source
IPSynth Tactic Synthesis	20 tasks (4–21 methods, 1–10 classes per task)	Security tactic implementation (JAAS authentication)	Java	Shokri et al. [5]

Dataset	Size	Scope	Language	Source
Refactoring Oracle	703 pure refactoring instances from 10 projects	Method-level refactoring (6 types: Extract/Move/Inline Method + compounds)	Java	MANTRA (Xu et al.) [9]
Fowler Benchmark	61 refactoring types + 53 real scenarios from ANTLR4/JUnit	Refactoring type coverage across Fowler's full catalog	Java (translated from JS)	Piao et al. [34]
LLM4Refactoring	180 real-world refactorings from 20 projects + 102 unit tests	LLM refactoring opportunity identification and solution recommendation (9 types)	Java	Liu et al. [33]
SO QA-AT Corpus	1,165 labeled posts (4,195 verified QA-AT instances)	Practitioner knowledge on tactic–quality attribute relationships	N/A (text)	Bi et al. [22]
AIDev Refactoring	15,451 refactoring instances across 12,256 PRs from 1,613 repositories	AI agent refactoring behavior in real-world open-source projects	Java	Horikawa et al. [37]
DePalma Refactoring	40 Java files x 8 quality attributes (320 trials)	LLM refactoring across quality attributes	Java	DePalma et al. [32]

Several observations stand out:

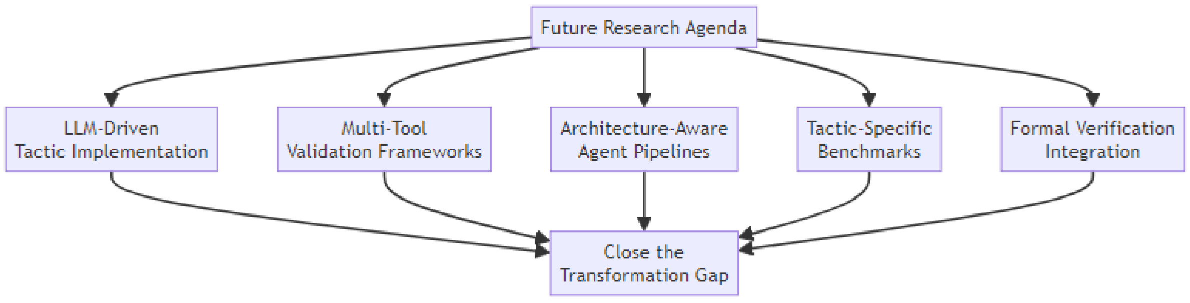
- **Every dataset is Java-only** (or language-independent text). There are no code-level datasets in Python, JavaScript, Go, Rust, or C++.
- **The largest dataset** (AIDev, 15,451 instances) captures *what agents have already done*, not what they *should* do — it is observational, not prescriptive.
- **The only tactic-level dataset** (IPSynth, 20 tasks) is restricted to a single tactic

type (authentication) in a single framework (JAAS), and is too small for statistical analysis.
- **No dataset includes architecture-level before/after pairs** with ground truth indicating which tactic was applied, the intended quality improvement, and verification criteria.

Creating such a dataset — labeled systems before and after tactic implementation, covering multiple languages and tactic types — is itself a significant research contribution waiting to be made.

9.4 Future Research Agenda

Based on the gaps identified above, we propose five concrete research directions that collectively address the transformation gap and its surrounding challenges.


Future Research Agenda
LLM-Driven Tactic Implementation
Multi-Tool Validation Frameworks
Architecture-Aware Agent Pipelines
Tactic-Specific Benchmarks
Formal Verification Integration
Close the Transformation Gap

9.4.1 LLM-Driven Architectural Tactic Implementation

The most direct contribution is to build and evaluate systems that use LLMs to implement architectural tactics in real codebases. This requires solving several sub-problems simultaneously:

- **Architecture context injection:** Before the LLM generates any code, it must receive a structured representation of the system's current architecture — dependency graphs, module boundaries, coupling/cohesion profiles, identified design smells. This is analogous to MANTRA's RAG component [9] but elevated to the architectural level.
- **Multi-file transformation:** Unlike method-level refactoring, tactic implementation inherently requires coordinated changes across multiple files. The system must generate a transformation plan (which files to modify, in what order, with what changes) and execute it atomically.
- **Tactic-specific prompting:** The instruction given to the LLM must encode the tactic's definition, its expected structural effect, and its applicability conditions. Piao et al.'s finding that rule-based instructions outperform descriptive ones [34] suggests that formalizing tactics as structured rules (pre-conditions, transformation steps, post-conditions) may be more effective than natural language descriptions.
- **Modifiability tactics as the starting point:** The 15 modifiability tactics catalogued by Bogner et al. [18] — organized into Increase Cohesion (Split Module, Maintain Semantic Coherence), Reduce Coupling (Use an Intermediary, Restrict Dependencies, Abstract Common Services, Encapsulate), and Defer Binding Time — provide a concrete, well-defined starting catalog.

A successful system in this space would demonstrate, for the first time, that LLMs can implement named architectural tactics with measurable quality improvement and verified behavior preservation.

9.4.2 Multi-Tool Validation Frameworks

The tool disagreement problem (less than 0.4% inter-tool agreement [28]) demands a validation framework that synthesizes evidence from multiple complementary tools rather than relying on any single one. Such a framework should include:

- **Multiple static analysis tools:** Combine Radon (Python-specific: Maintainability Index, cyclomatic complexity, Halstead metrics), SonarQube (broad rule coverage, technical debt estimation), Pylint (PEP 8 compliance, design checks), and architecture-specific analyzers (dependency graphs, coupling/cohesion at the module level).
- **Statistical rigor:** Use appropriate statistical tests for before/after comparison: Cliff's Delta for effect size (non-parametric, appropriate for non-normal metric distributions), Wilcoxon signed-rank test for paired comparisons, and Benjamini-Hochberg FDR correction for multiple comparisons — as Horikawa et al. demonstrate [37].
- **Metric-paradox awareness:** Track multiple metrics simultaneously and interpret them in the context of the specific tactic's intended effect, following Kim et al.'s finding that single-metric evaluation is misleading [38].

The goal is not merely to report "metric X improved" but to provide convergent evidence from multiple independent tools that the tactic implementation achieved its architectural intent.

9.4.3 Architecture-Aware Agent Pipelines

Current AI coding agents operate without architectural awareness, treating each code change in isolation [37]. The next generation of agents should understand system architecture *before* making changes:

- **Architecture-as-context:** Feed dependency graphs, module boundaries, quality profiles, and design smell reports to the LLM as structured input. This transforms the agent from a code-level responder into an architecture-level reasoner.
- **Proactive tactic selection:** Instead of waiting for a developer to request a specific change, the agent should analyze the system's architectural health, identify opportunities for tactic application, and propose tactic-level improvements — moving from "tactical cleanup" to "strategic architecture improvement" [37].
- **Multi-agent collaboration:** MANTRA's three-agent architecture (Developer, Reviewer, Repair) [9] provides a proven blueprint. For architecture-level work, this could be extended to include an Architect Agent (analyzes system architecture and selects tactics), a Developer Agent (generates code changes), a Reviewer Agent (verifies correctness using static analysis and tests), and a Repair Agent (fixes issues identified by the Reviewer).

9.4.4 Tactic-Specific Benchmark Creation

The absence of architecture-level transformation benchmarks is a bottleneck for the entire field. Creating such benchmarks requires:

- **Labeled before/after systems:** Real or synthetic codebases captured before and after a known tactic was implemented, with ground truth labels indicating the tactic type, the files affected, and the intended quality improvement.
- **Multi-language coverage:** Especially Python, which is underrepresented in all existing datasets despite being one of the most widely used languages for backend systems.
- **Verification criteria:** For each benchmark entry, a specification of how to verify that the tactic was correctly implemented — structural checks (does the intermediary class exist? are the original direct dependencies removed?), behavioral checks (do all tests pass?), and quality checks (did coupling decrease? did cohesion increase?).
- **Scalable construction:** Since manual benchmark creation is expensive (Liu et al. report 42 person-days for 180 instances [33]), explore semi-automated approaches: mine open-source repositories for commits that implement known tactics (using commit message analysis and structural diff patterns), then manually validate a subset.

9.4.5 Formal Verification Integration

The 7.4% unsafe transformation rate reported by Liu et al. [33] is acceptable for developer-reviewed suggestions but not for autonomous pipelines operating at scale. Bridging LLM output with lightweight formal methods could provide stronger guarantees:

- **AST-based structural verification:** After the LLM generates a transformation, parse the before and after versions into abstract syntax trees and verify that the structural changes match the tactic's expected pattern (e.g., for "Use an Intermediary": a new class was created, direct dependencies were replaced with dependencies through the new class, no direct dependencies remain).
- **Refinement calculus for behavioral equivalence:** Use program refinement techniques to verify that the transformed program is a valid refinement of the original — meaning it preserves all externally observable behaviors while allowing internal structural changes.
- **IPSynth's "correct by construction" paradigm:** IPSynth demonstrates that formal approaches (FSpec models, SMT solving) can achieve 85% semantic correctness on tactic synthesis [5]. A hybrid approach — using the LLM for generative code production and formal methods for verification — could combine the flexibility of LLMs with the guarantees of formal techniques.

9.5 Conclusion

This study guide has traced a path through six decades of software engineering research, from the foundational recognition that software architecture is the primary determinant of system quality [7], [10] to the modern possibility that large language models could automate the implementation of architectural design decisions.

The journey followed a logical arc:

1. **Software architecture foundations** established that architectural decisions — not individual lines of code — determine how maintainable, modifiable, and testable a system will be.
2. **Quality and maintainability models** (ISO/IEC 25010 and its predecessors) provided a standardized vocabulary for measuring software quality, decomposing "maintainability" into modularity, analysability, modifiability, reusability, and testability.
3. **Architectural tactics** gave architects a catalog of design decisions (Split Module, Use an Intermediary, Abstract Common Services, Restrict Dependencies) that systematically target specific quality attributes — the "building blocks" of quality-driven design [1].
4. **Architecture erosion** revealed that even well-designed systems degrade over time, with 83.8% of practitioners reporting quality decline [4], and that detecting erosion does not guarantee its repair [8].
5. **Assessment methods** showed that static analysis tools can measure maintainability, but with significant inter-tool disagreement (less than 0.4% [28]) and metric-dependent results [38], demanding multi-tool, multi-metric evaluation frameworks.
6. **LLMs for code refactoring** demonstrated that automated code transformation is feasible at the method level (82.8% success with multi-agent pipelines [9]) but remains confined to local, syntactic changes rather than architectural restructuring [37].
7. **Challenges** showed that context blindness, hallucinations, the complexity gap, tool disagreement, and the detection-remediation gap are real but not insurmountable barriers.

The field is at an inflection point. The individual components needed for automated architectural tactic implementation — tactic catalogs, LLM code generation, multi-agent pipelines, static analysis verification, iterative feedback loops — all exist. What does not yet exist is a system that combines them: one that takes a tactic specification, understands the target system's architecture, generates the multi-file transformation, verifies correctness, and measures quality improvement.

The **transformation gap** — the disconnect between tactic detection and LLM-based code refactoring — is the key unsolved problem. On one side, we know *what* to do (decades of architectural tactics research have produced detailed catalogs and quality attribute mappings). On the other side, we know *how to transform code* (LLMs with multi-agent verification can produce reliable method-level changes). Bridging this gap — connecting architectural *intent* with automated *implementation* — would advance both software engineering research (by providing the first empirical evidence of automated tactic implementation) and practice (by giving developers a tool that not only detects architectural problems but fixes them).

The research agenda outlined in this chapter — LLM-driven tactic implementation, multi-tool validation, architecture-aware agents, tactic-specific benchmarks, and formal verification integration — provides a roadmap for closing this gap. Future work in any of these directions would contribute meaningfully to the field. Work that addresses several of them simultaneously — as this thesis aims to do — has the potential to open an entirely new category of automated software architecture improvement.

Review questions.

1. Describe the "transformation gap" in your own words. Why have the tactic detection and LLM refactoring communities not yet converged?
2. Choose one of the seven specific research gaps from Section 9.2 and design a study that would address it. What data would you need? What metrics would you use? What would constitute a successful outcome?
3. Why is the IPSynth dataset (20 tasks) insufficient for drawing generalizable conclusions about automated tactic implementation? What properties should a successor dataset have?
4. Compare and contrast two of the five future research directions (Section 9.4). Which would have the highest impact if solved? Which is the most feasible with current technology? Justify your answers.
5. The conclusion states that the field is at an "inflection point." Do you agree? What developments (in LLMs, in static analysis, in software architecture) would need to occur for automated tactic implementation to become practical?

[1] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 4th ed. Addison-Wesley, 2021.

[2] M. Fowler, *Refactoring: Improving the design of existing code*, 2nd ed. Addison-Wesley, 2018.

[3] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, 1995.

[4] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding architecture erosion: The practitioners' perceptive," in *2021 IEEE/ACM 29th international conference on program comprehension (ICPC)*, IEEE, 2021, pp. 311–322.

[5] S. E. Shokri and R. Khatchadourian, "IPSynth: Interprocedural program synthesis for software architecture recovery and architectural tactics detection," in *arXiv preprint arXiv:2403.10836*, 2024. Available: https://arxiv.org/abs/2403.10836

[6] P. Haindl and G. Weinberger, "Does ChatGPT help novice programmers write better code? Results from static code analysis," *IEEE Access*, vol. 12, pp. 114547–114558, 2024, doi: 10.1109/ACCESS.2024.3445432.

[7] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992, doi: 10.1145/141874.141884.

[8] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: A case study," *Software: Practice and Experience*, vol. 41, no. 1, pp. 63–86, 2011.

[9] Y. Xu, F. Lin, J. Yang, T.-H. Chen, and N. Tsantalis, "MANTRA: Enhancing automated method-level refactoring with contextual RAG and multi-agent LLM collaboration," in *arXiv preprint arXiv:2503.14340*, 2025. Available: https://arxiv.org/abs/2503.14340

[10] D. Garlan and M. Shaw, "An introduction to software architecture," *Advances in software engineering and knowledge engineering*, vol. 1, no. 3, 1993.

[11] N. B. Harrison and P. Avgeriou, "How do architecture patterns and tactics interact? A model and annotation," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1735–1758, 2010.

[12] R. E. Al-Qutaish, "Quality models in software engineering literature: An analytical and comparative study," *Journal of American Science*, vol. 6, no. 3, pp. 166–175, 2010.

[13] A. B. Al-Badareen, M. H. Selamat, M. A. Jabar, J. Din, and S. Turaev, "Software quality models: A comparative study," in *International conference on software engineering and computer systems (ICSECS)*, Springer, 2011, pp. 46–55. doi: 10.1007/978-3-642-22170-5_4.

[14] "ISO/IEC 25010 — Systems and software engineering: Software product Quality Requirements and Evaluation (SQuaRE) — Product quality model." https://iso25000.com/en/iso-25000-standards/iso-25010, 2023.

[15] A.-J. Molnar and S. Motogna, "A study of maintainability in evolving open-source software," in *International conference on evaluation of novel approaches to software engineering*, Springer, 2020, pp. 261–282.

[16] J. Visser, S. Rigal, R. van der Leij, P. van Eck, and G. Wijnholds, *Building maintainable software, Java edition: Ten guidelines for future-proof code.* O'Reilly Media, 2016.

[17] S. Kim, D.-K. Kim, L. Lu, and S. Park, "Quality-driven architecture development using architectural tactics," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1211–1231, 2009.

[18] J. Bogner, S. Wagner, and A. Zimmermann, "Using architectural modifiability tactics to examine evolution qualities of service- and microservice-based systems," *Computer Science – Research and Development*, vol. 34, pp. 187–237, 2019, doi: 10.1007/s00450-019-00402-z.

[19] M. Kassab, M. Mazzara, J. Lee, and G. Succi, "Software architectural patterns in practice: An empirical study," *Innovations in Systems and Software Engineering*, vol. 14, no. 4, pp. 263–271, 2018, doi: 10.1007/s11334-018-0319-4.

[20] G. Márquez, H. Astudillo, and R. Kazman, "Architectural tactics in software architecture: A systematic mapping study," *Journal of Systems and Software*, vol. 197, p. 111558, 2022, doi: 10.1016/j.jss.2022.111558.

[21] Z. Rahmati and M. Tanhaei, "Ensuring software maintainability at software architecture level using architectural patterns," *AUT Journal of Mathematics and Computing*, vol. 2, no. 1, pp. 81–102, 2021, doi: 10.22060/ajmc.2021.19232.1044.

[22] T. Bi, P. Liang, A. Tang, and X. Xia, "Mining architecture tactics and quality attributes knowledge in Stack Overflow," *Journal of Systems and Software*, vol. 180, p. 111005, 2021, doi: 10.1016/j.jss.2021.111005.

[23] L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A tool-based perspective on software code maintainability metrics: A systematic literature review," *Scientific Programming*, vol. 2020, p. 8840389, 2020, doi: 10.1155/2020/8840389.

[24] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Conference on software maintenance*, IEEE, 1992, pp. 337–344.

[25] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE–2, no. 4, pp. 308–320, 1976, doi: 10.1109/TSE.1976.233837.

[26] M. H. Halstead, *Elements of software science.* Elsevier, 1977.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994, doi: 10.1109/32.295895.

[28] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, p. 111575, 2023.

[29] H. Cui, M. Xie, T. Su, C. Zhang, and S. H. Tan, "An empirical study of false negatives and positives of static code analyzers from the perspective of historical issues," in *arXiv preprint arXiv:2408.13855*, 2024. Available: https://arxiv.org/abs/2408.13855

[30] S. Nocera, D. Fucci, and G. Scanniello, "Dealing with SonarQube Cloud: Initial results from a mining software repository study," in *2025 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, 2025. doi: 10.1109/ESEM64174.2025.00035.

[31] S. Martinez, L. Xu, M. Elnaggar, and E. A. AlOmar, "Software refactoring research with large language models: A systematic literature review," *Journal of Systems and Software*, p. 112762, 2025, doi: 10.1016/j.jss.2025.112762.

[32] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, "Exploring ChatGPT's code refactoring capabilities: An empirical study," *Expert Systems with Applications*, vol. 249, p. 123602, 2024.

[33] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "Exploring the potential of general purpose LLMs in automated software refactoring: An empirical study," *Automated Software Engineering*, vol. 32, no. 1, 2025, doi: 10.1007/s10515-025-00500-0.

[34] Y. C. K. Piao, J. C. Paul, L. Da Silva, A. M. Dakhel, M. Hamdaqa, and F. Khomh, "Refactoring with LLMs: Bridging human expertise and machine understanding," in *arXiv preprint arXiv:2510.03914*, 2025. Available: https://arxiv.org/abs/2510.03914

[35] A. Cordeiro, J. Ribeiro, *et al.*, "An empirical study on LLM-based agents for automated bug fixing and refactoring," in *arXiv preprint arXiv:2411.02320*, 2024. Available: https://arxiv.org/abs/2411.02320

[36] J. C. Gonçalves and M. Maia, "An empirical study on the effectiveness of iterative LLM-based improvements for static analysis issues," in *Brazilian symposium on software engineering (SBES)*, 2025. doi: 10.5753/sbes.2025.9964.

[37] K. Horikawa, H. Li, Y. Kashiwa, B. Adams, H. Iida, and A. E. Hassan, "Agentic refactoring: An empirical study of AI coding agents," in *arXiv preprint arXiv:2511.04824*, 2025. Available: https://arxiv.org/abs/2511.04824

[38] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014, doi: 10.1109/TSE.2014.2318734.

[39] Y. Ge, Q. Wang, H. Zhang, K. Miao, and Y. Li, "ArchTacRV: Detecting and runtime verifying architectural tactics in code," in *IEEE international conference on software analysis, evolution and reengineering (SANER)*, 2022. doi: 10.1109/saner53432.2022.00074.