INNOPOLIS UNIVERSITY — MS STUDY GUIDE, 2026

Automated Implementation of Architectural Tactics for Software Quality Improvement

A Comprehensive Study Guide

February 2026

Study Guide Overview

Ch 1: Motivation & Context
Why maintenance costs dominate

Ch 2: SA Foundations
Components, connectors, styles

Ch 3: Quality & Maintainability
ISO 25010, metrics, SIG model

Ch 4: Architectural Tactics
15 modifiability tactics catalog

Ch 5: Architecture Erosion
Detection-remediation gap

Ch 6: Assessment Methods
Metrics, tools, agreement crisis

Ch 7: LLMs for Refactoring
MANTRA, prompts, pipelines

Ch 8: Challenges
Technical, measurement, practical

Ch 9: Research Gaps
The transformation gap

40+ papers reviewed across architectural tactics, maintainability, and LLM-based code transformation

CHAPTER 1

Motivation & Context

Why software maintenance dominates lifecycle costs and how architectural decisions determine maintainability

The Software Maintenance Crisis

The root cause is rarely individual lines of code — it is **decisions at the architectural level** that determine how easily a system absorbs change.

60–80%

Maintenance share of total lifecycle cost

[Bass et al., 2021]

75%

Development effort spent on refactoring

[Fowler, 2018]

83.8%

Practitioners reporting quality degradation from erosion

[Li et al., 2021]

"Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the ***load-bearing walls*** of a system, maintainers can better understand the ramifications of changes."

— Garlan & Perry, 1995

The Knowledge Barrier

Tactics Require Deep Knowledge

- Architectural tactics are easy to state but hard to implement in real codebases
- Require **inter-procedural changes** spanning multiple classes and packages
- ChatGPT: 95% syntax-correct but only **5% semantically correct** on tactic synthesis [Shokri et al., 2024]

Detection ≠ Remediation

- Rosik et al. (IBM): **0 of 9** detected violations were fixed by developers
- Barriers: risk of ripple effects, time pressure, legacy entanglement
- Developers **knew** about problems but chose not to fix them

The Vicious Cycle

Technical debt causes erosion → erosion generates more debt → architecture degrades further

[Li et al., 2021]

The Promise

LLMs can bridge the gap if placed in a **structured pipeline** with external verification

MANTRA: **82.8%** success rate with multi-agent pipeline vs 8.7% standalone

CHAPTER 2

Software Architecture Foundations

Components, connectors, configurations, and the architectural styles that shape system quality

What Is Software Architecture?

Perry & Wolf (1992)
Architecture = Elements + Form + Rationale

Garlan & Shaw (1993)
Components + Connectors + Configurations

Bass, Clements & Kazman
Structures needed to reason about the system

Architecture captures intent, not just structure — rationale is a first-class citizen

Key Architectural Styles

- **Pipes-and-Filters:** Data flows through stages
- **Layered:** Hierarchical dependency rules
- **Event-Driven:** Components via events
- **Repository:** Shared data store
- **Client-Server:** Requestors and providers

KWIC case study: same problem in 4 styles yields different modifiability trade-offs

CHAPTER 3

Quality & Maintainability

From McCall (1977) to ISO/IEC 25010 — how we define, decompose, and measure software quality

ISO/IEC 25010: Maintainability

Maintainability is one of **8 quality characteristics**. It decomposes into 5 sub-characteristics:

Modularity
Changes to one component have minimal impact on others

Reusability
Assets can be used in building other systems

Analysability
Effectiveness of assessing change impact

Modifiability
Can be modified without degradation — **thesis focus**

Testability
Test criteria can be established and tests executed

Key Finding: Hotspot Concentration
10% of packages contain 80% of maintainability issues [Molnar & Motogna, 2020]

SIG Maintainability Model
Visser's 10 guidelines with 1-5 star rating; issue resolution is **2x faster** in 4-star vs 2-star systems

CHAPTER 4

Architectural Tactics

Design decisions that influence the achievement of a quality attribute response

What Are Architectural Tactics?

Definition: A design decision that influences the achievement of a quality attribute response

— Bass, Clements & Kazman, 2021

Design Hierarchy

- **Style:** Overall system organization (e.g., Layered)
- **Pattern:** Recurring solution (e.g., MVC)
- **Tactic:** Targeted quality decision (e.g., Use Intermediary)
- **Technique:** Implementation detail (e.g., Adapter class)

Kassab et al.: **63%** of projects modify patterns with tactics

Stimulus → Tactic → Response

Stimulus: A change request arrives

Tactic: Use an Intermediary to decouple

Response: Change is confined to fewer modules

Tactic Research Landscape

- Marquez (2022): 91 studies surveyed
- **71%** don't describe identification method
- Detection: NLP/BERT, ML classifiers, Archie, ArchEngine
- The rigor gap: most studies are qualitative

Modifiability Tactics Catalog

Increase Cohesion

- Split Module
- Increase Semantic Coherence
- Abstract Common Services
- Encapsulate
- Restrict Dependencies

Reduce Coupling

- Use an Intermediary
- Use Encapsulation
- Restrict Dependencies
- Abstract Common Services

Defer Binding

- Component Replacement
- Publish-Subscribe
- Configuration Files
- Polymorphism
- Protocol Binding
- Runtime Registration

SOA: Reduce Coupling dominates (49%) [Bogner, 2019]

Microservices: Defer Binding dominates (52%) [Bogner, 2019]

Harrison (2010): 7 interaction types between patterns & tactics

CHAPTER 5

Architecture Erosion & Drift

Why detection alone does not solve architectural degradation

Erosion vs. Drift

Erosion = Breaking the Rules

Explicit constraints violated — e.g., controller directly accessing the database layer

Drift = Losing the Map

Gradual divergence — no single rule broken, but the system drifts from intended design

Li et al. (2021) — Four Perspectives

- **Violation:** Implementation violates intended architecture
- **Structure:** Cyclic dependencies, god classes accumulate
- **Quality:** Maintainability, performance degrade
- **Evolution:** Architecture resists change

Causes of Erosion

Architecture violation — 24.7%

Evolution issues — 23.3%

Technical debt — 17.8%

Knowledge vaporization — 15.1%

35 tools cataloged for erosion detection, but 82.2% of studies are academic — developers don't use them in practice

IBM Dublin Case Study [Rosik et al., 2011]

2-year longitudinal study of DAP 2.0 (28,500 LOC) using Reflexion Modelling

9

Divergent edges detected

0

Violations fixed

Why Violations Persist

- **Risk aversion:** "Fixing minor issues might cause larger ones"
- **Cost-benefit:** Minor violations perceived as not worth the effort
- **Time pressure:** Architectural fixes never "in scope"

Hidden Violations

5 of 8 convergent edges contained hidden divergent relationships

Worst case: 1 edge masked **41 of 44** inconsistent relationships

> *"If performance can be gained by breaking an architectural design, then this can sometimes be acceptable... time pressure is probably the main factor."*
>
> — IBM developer

Key insight: Continuous monitoring needed, not batch detection

The Detection-Remediation Gap

Detection exists but is underused — 35 tools, but 82.2% academic-only

Detection does not lead to action — 0/9 violations fixed at IBM

Knowledge exists but is fragmented — 21% of QA-tactic relationships are undocumented

Current Workflow (Manual)

Detection → Manual Analysis → Manual Design → Manual Implementation

Each step is a barrier; each barrier defers the fix

Proposed Workflow (Automated)

Detection → LLM: Select Tactic → LLM: Implement → Static Analysis Validation

How LLMs Address Barriers

- **Risk reduced:** Changes validated by testing before deployment
- **Cost reduced:** No human architect needed for solution design
- **Time reduced:** Automated, runs in CI/CD on every commit

Stack Overflow mining (Bi et al., 2021): **4,195 posts** on tactics, but only 11% discuss applying tactics to existing systems

CHAPTER 6

Maintainability Assessment

Metrics, tools, and the surprising disagreement between static analysis tools

Key Maintainability Metrics

Maintainability Index (MI)

MI = 171 - 5.2·ln(V) - 0.23·G - 16.2·ln(LOC)

>85 = A (good), 65-85 = B, <65 = C (bad)

Cyclomatic Complexity (CC)

Decision points + 1; Visser recommends CC ≤ 5

1-10 = Low, 11-20 = Moderate, 21-50 = High, >50 = Very High

Halstead Volume

V = N · log2(n) — information content of the program

Used in MI formula; higher = harder to maintain

CK Object-Oriented Suite

CBO — Coupling Between Objects (#1 cited metric, 18 studies)
RFC — Response For a Class (#2 cited, 17 studies)
WMC — Weighted Methods per Class
LCOM — Lack of Cohesion in Methods
DIT/NOC — Inheritance depth/breadth

Ardito et al. (2020): **174 distinct metrics** cataloged, but 75% appear in only a single paper. Field has converged on ~15 core metrics.

The Agreement Crisis

[Lenarduzzi et al., 2023] — 6 tools × 47 Java projects

< 0.4%

Overall inter-tool detection agreement

Tool Precision

- CheckStyle: **86%** (mostly formatting)
- FindBugs: **57%**
- PMD: **52%**
- SonarQube: **18%** (broadest but least precise)

> *"There is no silver bullet that is able to guarantee source code quality assessment on its own."*
>
> — Lenarduzzi et al., 2023

Implications

- Any improvement measured by a **single tool** may not be confirmed by another
- Cui et al. (2024): 35.7% of FN/FP from **missing cases**
- Issue count ≠ debt reduction (ratio as low as 0.71)

Solution: Multi-tool validation — Radon + SonarQube + Pylint + custom architecture analysis

CHAPTER 7

LLMs for Code Refactoring

From code completion to code transformation — capabilities, pipelines, and the architecture gap

The LLM Revolution in SE

From syntax-level code completion (2021) to semantic-level code transformation (2023+)

What LLMs Handle Well

- Rename Method/Variable
- Extract Method
- Simplify Conditionals
- Remove Dead Code
- Inline Variable

Local, single-file, pattern-based

What LLMs Struggle With

- Extract Class (0% recall with generic prompts)
- Move Method (cross-file)
- Split Class
- Extract Superclass

Structural, cross-file, design-judgment

Key Stats

97.2%
Behavior preservation [DePalma]

86.7%
Recall with optimized prompts [Liu]

3.7%
Agentic refactorings >1 file

Prompt Engineering for Refactoring

[Piao et al., 2025] — 5 strategies across all 61 Fowler refactoring types

Zero-Shot — Just the refactoring name	47.5% / 91.8%
Two-Shot — Two worked examples	57.4% / 95.1%
Step-by-Step — Procedural instructions	**83.6% / 100%**
Rule-based — Detection heuristics	**80.3% / 100%**
Objective Learning — High-level goal only	29–36% / 31–38%

Format: GPT-4o-mini / DeepSeek-V3 success rates

Generic → Subcategory prompt boosts success from 15.6% to 86.7%

[Liu et al., 2025]

Key Insight

Rule-based instructions (pre/post conditions) outperform descriptive ones — LLMs follow mechanical rules better than natural language

MANTRA: Multi-Agent Refactoring

[Xu et al., 2025] — The most significant advance in LLM-based refactoring to date

Developer Agent
Generates code via Context-Aware RAG

Reviewer Agent
Validates via SE tools (most critical)

Repair Agent
Iterative fixes via Reflexion

82.8%
MANTRA success

8.7%
Raw GPT baseline

9.5x
Improvement

Ablation: What Matters Most

- Without Reviewer: **-61.9%**
- Without Repair: **-50.7%**
- Without RAG: **-40.7%**

Key principle: External verification via traditional tools is more valuable than additional LLM reasoning

Tool-Integrated Pipelines

SonarQube + LLM Loop [Goncalves, 2025]

Codebase → SonarQube Analysis → Issues List → LLM Prompt → Fix → Re-analyze → Repeat

- Best config: **81.3%** issue reduction
- Average: >58% across all configurations
- 5 iterations outperform 2 (diminishing returns after 3-4)
- Low temp (0.1) + zero-shot = most consistent

Caveat: 58.8% issue reduction = only 42.1% debt reduction (ratio 0.71). LLMs can create new issues while fixing others.

Agentic Refactoring [Horikawa, 2025]

15,451 instances from AIDev dataset — largest corpus

- 26.1% of agentic commits target refactoring
- Dominant: Rename (10.4%), Change Type (11.8%)
- Only **3.7%** touch more than one file
- Median design smell delta: **0.00**
- 86.9% of PRs merged despite low impact

"Agentic coding tools serve as ***incremental cleanup partners***, not software architects."

— Horikawa et al., 2025

The Code-Level vs. Architecture-Level Gap

No existing work uses LLMs to implement architectural tactics.

Code-Level (Current)

- **Scope:** Single method or class
- **Context:** Local (<300 LOC)
- **Examples:** Extract Method, Rename
- **Files touched:** 1 (96.3%)
- **Design judgment:** Minimal
- **Best success:** 82.8% (MANTRA)

Architecture-Level (Thesis Gap)

- **Scope:** Multiple modules/packages
- **Context:** System-wide (project structure)
- **Examples:** Split Module, Use Intermediary
- **Files touched:** 5-15+
- **Design judgment:** Substantial
- **Best success:** Unknown — no data

Why the gap: Context window limits, no architecture-aware tools, training data bias (most commits are code-level)

The opportunity: Patterns from MANTRA + SonarQube pipeline can be extended from code-level to architecture-level

CHAPTER 8

Challenges & Limitations

Five core technical challenges and the measurement problems that complicate evaluation

Five Core Technical Challenges

HIGH

Context Blindness

LLMs see one file, tactics span many. RAG helps 40.7%.

CRITICAL

Hallucinations

95% syntax-correct but only 5% semantically correct on tactics.

HIGH

Token Limits

Performance degrades beyond ~300 LOC. Tactics need 10K+.

HIGH

Complexity Gap

0% Extract Class recall; agents are "cleanup partners."

MEDIUM

Non-Determinism

Same prompt → different outputs. Temp=0 helps.

Measurement Challenges

- Tool disagreement: <0.4%
- No tactic-specific metrics
- Java-dominated datasets
- Small benchmarks (IPSynth: 20 tasks)

Practical Challenges

- Detection-remediation gap (0/9 fixed)
- Developer resistance to automated changes
- Behavior preservation vs. architecture improvement
- Strict preservation may be inappropriate for tactic-level

CHAPTER 9

Research Gaps & Future Directions

The transformation gap and the five-point research agenda

The Transformation Gap

Two mature research streams exist in **almost complete isolation**

Stream 1: Tactic Detection

- Marquez: 91 studies surveyed
- ArchTacRV: ML-based detection
- IPSynth: 85% via program synthesis
- Bi et al.: 4,195 SO posts mined

GAP

Stream 2: LLM Refactoring

- MANTRA: 82.8% success
- Liu et al.: 86.7% recall
- Piao et al.: 61 types tested
- Horikawa: 15,451 instances

Thesis Contribution: Tactic Detection → **LLM Tactic Implementation** → Quality Measurement

IPSynth: 85% success but uses SMT solver, not LLMs. ChatGPT: only 5%

MANTRA: 82.8% but method-level only. No architecture-level data exists.

71% of tactic studies don't describe identification method [Marquez, 2022]

Seven Specific Research Gaps

#	Gap	Evidence
1	70% lack tactic identification methods	65 of 91 studies [Marquez, 2022]
2	Design rationale doesn't trace to code	Detected violations not remediated [Rosik, 2011]
3	No cost-benefit quantification for tactics	Qualitative mappings only [Bogner, 2019]
4	LLM + static analysis integration is ad hoc	No standardized framework exists
5	No Python architectural tactic benchmarks	All datasets Java-only
6	Agents don't plan architecturally	"Cleanup partners" not "architects" [Horikawa, 2025]
7	No formal verification of LLM refactorings	7.4% unsafe rate [Liu, 2025]; IPSynth SMT only

Future Research Agenda

1. LLM-Driven Tactic Implementation

Architecture context injection, multi-file transformation, tactic-specific prompting, 15 modifiability tactics as starting catalog

2. Multi-Tool Validation Frameworks

Radon + SonarQube + Pylint + architecture analysis with Cliff's Delta + Wilcoxon + Benjamini-Hochberg

3. Architecture-Aware Agent Pipelines

Architect + Developer + Reviewer + Repair agents; proactive tactic selection based on system health

4. Tactic-Specific Benchmarks

Labeled before/after systems, multi-language (especially Python), verification criteria included

5. Formal Verification Integration

AST-based structural verification, refinement calculus, hybrid LLM + SMT approach

The field is at an **inflection point**. All components exist individually — tactic catalogs, LLM code generation, multi-agent pipelines, static analysis verification. What doesn't exist is a system that **combines them all**.

Key Takeaways

Architecture determines maintainability

60-80% of costs are maintenance; load-bearing walls metaphor

Tactics are the building blocks

15 modifiability tactics in 3 categories; 63% of projects modify patterns with tactics

Detection ≠ Remediation

0/9 violations fixed at IBM; 82.2% academic-only tools

Never trust a single tool

<0.4% inter-tool agreement; multi-tool validation essential

LLMs work with pipelines

82.8% (MANTRA) vs 8.7% (standalone); verification is the critical component

Prompt specificity matters enormously

15.6% → 86.7% with subcategory prompts; rule-based outperforms descriptive

The gap is architecture-level

96.3% of agentic refactorings touch 1 file; no LLM implements tactics

The transformation gap can be bridged

All components exist individually — combining them is the research frontier

Thank You

Automated Implementation of Architectural Tactics
for Software Quality Improvement

Innopolis University — MS Study Notes, 2026

Based on 40+ papers across SA, maintainability, and LLM research