FORGE EVOLVE: A Tool-Agnostic AI Code Modernization Engine with Behavioral Equivalence Guarantees

A FORGE OS Subsystem Specification

577 Industries R&D Lab
577 Industries Incorporated
research@577industries.com

Abstract

Enterprise software systems worldwide carry an estimated $2.4 trillion in annual legacy maintenance costs, with 73% of organizations reporting critical technical debt in mission-essential codebases. The developers who built these systems are retiring, documentation is sparse or nonexistent, and manual rewrite programs—which consume 60% of IT budgets and fail at a 70% rate—cannot scale to meet the modernization imperative. Meanwhile, AI-powered coding tools such as Claude Code, OpenAI Codex, Google Gemini, and Cursor have demonstrated remarkable capabilities in code analysis and transformation, yet no single tool excels across all modernization tasks, languages, and architectural patterns.

This paper presents FORGE EVOLVE, the code modernization subsystem of FORGE OS—the Agent-Legible Operating System for enterprise AI. FORGE EVOLVE serves as the "DNA" of the FORGE OS platform: the prerequisite transformation engine that converts legacy codebases into modern, well-documented, AI-ready software, thereby unlocking every other FORGE OS subsystem. We introduce five integrated capabilities: (1) **Multi-Agent Tool Orchestration**, which dispatches modernization tasks to the optimal AI coding tool based on task-specific feature vectors; (2) **Behavioral Equivalence Verification**, combining property-based testing, semantic differencing, and automated test generation to guarantee functional preservation; (3) **Automated Business Rule Extraction**, using AST analysis coupled with LLM interpretation to surface implicit domain logic as structured RDF triples; (4) **Complexity-Aware Migration Planning**, employing graph-based decomposition with risk scoring to sequence migration units optimally; and (5) **Cross-Language Abstract Representation** (CLAR), a language-agnostic intermediate form that decouples source analysis from target generation.

Experimental evaluation across five legacy language families (COBOL, Fortran, Ada, Visual Basic 6, and legacy Java) demonstrates behavioral equivalence pass rates exceeding 99% on module-level transformations, business rule extraction F1 scores above 0.85, and 60–70% velocity improvements over manual migration baselines. All extracted knowledge flows into FORGE MEMORY's knowledge graph, cryptographic inventories feed FORGE QBIT's post-quantum migration queue, modernized control logic targets FORGE KINETIC's autonomous deployment, and clean codebases enable FORGE CORTEX's domain-specific fine-tuning—establishing FORGE EVOLVE as the foundational gateway to enterprise AI adoption.

1 Introduction

The global software estate is aging. Decades of incremental development have produced vast repositories of COBOL, Fortran, Ada, Visual Basic, and early-generation Java code that underpin financial transactions, defense logistics, healthcare records, and critical infrastructure (Khadka et al., 2014; Seacord et al., 2003). These legacy systems were built by engineers who are now approaching or past retirement age, embedding institutional knowledge into code that was never adequately documented. The result is a crisis of *technical debt at civilizational scale*: organizations cannot retire the systems because they cannot understand them, yet cannot maintain them because the expertise is disappearing.

1.1 The Legacy Code Crisis

The magnitude of the legacy modernization challenge is staggering. Industry analyses estimate that global enterprises spend $2.4 trillion annually maintaining legacy systems (Gartner, 2024), with 73% of organizations reporting that legacy technical debt directly impedes their ability to adopt modern technologies (Stripe, 2018). Approximately 60% of enterprise IT budgets are consumed by maintenance activities rather than innovation (Standish Group, 2020), creating a compounding cycle where the inability to modernize diverts resources that could fund modernization.

The human dimension compounds the technical challenge. The average COBOL programmer in the United States is over 55 years old, and the global population of Fortran-proficient engineers is declining at an estimated 8% per year (Micro Focus, 2020). When these developers retire, they take with them decades of undocumented domain knowledge—business rules encoded as cryptic variable names, edge cases handled by deeply nested conditional logic, and architectural decisions preserved only in institutional memory.

Manual rewrite programs have proven inadequate. Large-scale studies report that approximately 70% of manual legacy modernization projects fail to deliver on time, on budget, or with acceptable quality (Standish Group, 2020; Brodie & Stonebraker, 1995). These programs typically span 3–5 years, consume $10M–$100M+ in engineering costs, and introduce behavioral regressions that erode stakeholder confidence. The fundamental problem is that manual rewrites require developers to simultaneously understand legacy semantics (which are poorly documented), implement equivalent modern code (which demands different skills), and verify behavioral preservation (which is combinatorially expensive)—three cognitively demanding tasks that are difficult to execute in parallel.

1.2 The AI Code Modernization Opportunity

The emergence of AI-powered coding tools has created an unprecedented opportunity to automate legacy modernization. Tools such as Claude Code (Anthropic, 2024), OpenAI Codex (Chen et al., 2021), Google Gemini (Google, 2024), and Cursor (Cursor, 2024) have demonstrated remarkable capabilities in code understanding, generation, and transformation. These tools can analyze legacy code semantics, generate equivalent modern implementations, produce documentation, and even create test suites—tasks that previously required months of expert human effort.

However, no single AI tool excels at every modernization task. Our empirical analysis reveals significant performance variation across task types:

- **Claude Code** excels at deep semantic analysis, type-safe translation, and complex refactoring that requires understanding business context across multiple files.
- **Codex** performs best on batch translation tasks with well-defined input/output patterns, particularly for API migration and syntax transformation.
- **Gemini** leverages its large context window for physics-aware and domain-specific translation of scientific computing codebases, where understanding mathematical semantics is essential.
- **Cursor** provides interactive refinement capabilities ideal for UI/business logic separation and incremental modernization with developer collaboration.

This heterogeneity in tool capabilities creates a meta-optimization problem: given a legacy codebase with diverse modernization tasks, how should an organization select and orchestrate multiple AI tools to maximize quality while minimizing cost and time? Current practice relies on ad-hoc tool selection

by individual developers, leading to suboptimal assignments, inconsistent quality, and the inability to leverage each tool's comparative advantage.

1.3 FORGE EVOLVE: The DNA of FORGE OS

We introduce FORGE EVOLVE, the code modernization subsystem of FORGE OS (577 Industries, 2025a), designed to address the legacy modernization crisis through systematic, tool-agnostic AI orchestration. We position FORGE EVOLVE as the "DNA" of FORGE OS for a precise reason: just as DNA provides the foundational code that enables all biological functions, FORGE EVOLVE transforms the foundational software code that enables all other FORGE OS capabilities.

Until legacy code is modernized, organizations cannot effectively adopt any other FORGE OS subsystem:

- **FORGE MEMORY** cannot build knowledge graphs from undocumented COBOL spaghetti code—but it can ingest the structured business rules that FORGE EVOLVE extracts.
- **FORGE QBIT** cannot perform post-quantum cryptographic migration on systems whose cryptographic inventory is buried in Fortran subroutines—but it can act on the crypto inventories that FORGE EVOLVE surfaces.
- **FORGE KINETIC** cannot deploy autonomous systems built on legacy control logic—but it can target the modernized, well-documented control code that FORGE EVOLVE produces.
- **FORGE CORTEX** cannot fine-tune domain-specific models on poorly structured legacy codebases—but it can leverage the clean, documented code that FORGE EVOLVE generates as high-quality training data.

In this sense, FORGE EVOLVE is not merely another modernization tool—it is the *prerequisite* for enterprise AI adoption, the transformation engine that converts an organization's most valuable (and most neglected) software assets into the foundation for modern AI capabilities.

1.4 Contributions

The primary contributions of this work are sevenfold:

1. **Multi-Agent Tool Orchestration Framework** (Section 4.3): A principled approach to dispatching modernization tasks to the optimal AI coding tool based on task-specific feature vectors, achieving 23% higher quality scores than single-tool baselines while reducing cost by 31%.
2. **Behavioral Equivalence Verification Pipeline** (Section 4.5): An integrated pipeline combining property-based testing (Claessen & Hughes, 2000), semantic differencing, and automated test generation that achieves $>$99% behavioral equivalence pass rates on module-level transformations across five language families.
3. **Automated Business Rule Extraction** (Section 4.1): A hybrid AST-analysis and LLM-interpretation system that extracts implicit business rules from legacy code and represents them as structured RDF triples, achieving F1 scores $>$0.85 across financial, defense, and healthcare domains.
4. **Complexity-Aware Migration Planning** (Section 4.2): A graph-based decomposition algorithm with risk scoring that sequences migration units to minimize cascading failures and maximize early value delivery, with formal hardness results for optimal ordering (Theorem 2).
5. **Cross-Language Abstract Representation** (CLAR) (Section 4.2): A language-agnostic intermediate representation that decouples source analysis from target generation, enabling N-to-M language transformation through a single canonical form.
6. **Knowledge Preservation via FORGE MEMORY** (Section 5): A bidirectional integration protocol through which extracted business rules, architectural decisions, and migration rationale flow into FORGE MEMORY's Immutable Governance Object Model (IGOM) for permanent organizational knowledge preservation.
7. **FORGE OS Telemetry Integration** (Section 5): Emission of `MODERNIZATION_TASK`, `EQUIVALENCE_CHECK`, and `RULE_EXTRACTION` events into the unified `ForgeEvent` stream, enabling cross-subsystem observability and governance compliance.

1.5 Paper Organization

Section 2 reviews related work across legacy modernization, AI code generation, program analysis, software verification, knowledge extraction, and multi-agent orchestration. Section 3 formalizes the modernization problem with rigorous definitions and theoretical results. Section 4 presents the FORGE EVOLVE system architecture in detail across five core modules. Section 5 describes integration with FORGE OS subsystems. Section 6 covers implementation details and technology choices. Section 7 presents experimental evaluation across five language families. Section 8 provides domain-specific case studies in financial services, defense, healthcare, and enterprise IT. Section 9 discusses findings, limitations, and ethical considerations. Section 10 concludes with future directions.

2 Related Work

FORGE EVOLVE draws upon and extends six streams of research: legacy system modernization strategies, AI-powered code generation and transformation, program analysis and understanding, software verification and testing, knowledge extraction from code, and multi-agent orchestration.

2.1 Legacy System Modernization

The software engineering community has studied legacy modernization for over three decades, producing several canonical strategies. The *Chicken Little* approach (Brodie & Stonebraker, 1995) advocates incremental, module-by-module migration with a gateway layer that mediates between legacy and modern components during the transition period. While risk-reducing, Chicken Little incurs substantial gateway overhead and can extend migration timelines to 5–10 years for large systems. The *Big Bang* approach (Bisbal et al., 1999) replaces the entire legacy system at once, offering a clean architectural break but carrying extreme risk: if the replacement fails, there is no fallback. The *Butterfly* methodology (Wu et al., 1997) introduces a data migration strategy that creates a parallel modern system and gradually redirects data flows, minimizing operational disruption but requiring sophisticated data synchronization.

Modernization assessment frameworks provide structured evaluation criteria. The Gartner Application Modernization Framework (Gartner, 2024) classifies modernization options along a spectrum from encapsulate (wrap legacy APIs) through rehost, replatform, refactor, rearchitect, to rebuild and replace. Comella-Dorda et al. (2000) formalize modernization decision criteria based on technical quality, business value, and organizational readiness. Khadka et al. (2014) provide a comprehensive systematic literature review identifying 121 primary studies on legacy modernization, concluding that no single strategy dominates and that hybrid approaches are most effective.

FORGE EVOLVE adopts a *hybrid incremental* strategy most closely aligned with Chicken Little but enhanced with AI-driven analysis that dramatically accelerates the understanding phase. Unlike prior frameworks that assume human analysts will decompose the legacy system, FORGE EVOLVE automates decomposition through its Discovery Engine and Migration Planner, reducing the planning phase from months to days.

2.2 AI Code Generation and Transformation

The application of machine learning to code generation has advanced rapidly. OpenAI Codex (Chen et al., 2021) demonstrated that large language models pretrained on code repositories could generate functionally correct programs from natural language descriptions, achieving 28.8% pass@1 on the HumanEval benchmark. AlphaCode (Li et al., 2022) extended this to competitive programming, generating solutions at the level of median competitors. Claude Code (Anthropic, 2024) introduced agentic coding capabilities where the model can navigate file systems, run tests, and iteratively refine implementations. Google Gemini (Google, 2024) brought million-token context windows that enable whole-codebase understanding.

Code translation—the task of converting source code from one programming language to another—is directly relevant to legacy modernization. TransCoder (Rozière et al., 2020) pioneered unsupervised code translation using denoising auto-encoders trained on monolingual code corpora, achieving successful translation between C++, Java, and Python without parallel data. Pan et al. (2024) provide a comprehensive evaluation of LLM-based code translation, identifying systematic failure modes

including type system mismatches, library API incompatibilities, and semantic drift in numeric precision.

Agent-based coding systems represent the frontier of AI-assisted development. SWE-bench (Jimenez et al., 2024) established a benchmark for evaluating AI systems on real-world software engineering tasks drawn from GitHub issues. Devin (Cognition AI, 2024) demonstrated end-to-end autonomous software engineering on a subset of these tasks. Aider (Aider, 2024) and similar tools provide chat-based interfaces for iterative code modification with version control integration.

FORGE EVOLVE differs from these systems in three critical respects. First, it is *tool-agnostic*: rather than building a single AI coding agent, FORGE EVOLVE orchestrates multiple existing tools, selecting the optimal tool for each task. Second, it provides *formal behavioral equivalence guarantees* rather than relying solely on test passage or human review. Third, it operates at *enterprise scale*, handling multi-million-line codebases through systematic decomposition rather than addressing individual files or functions in isolation.

2.3 Program Analysis and Understanding

Static program analysis provides the foundational techniques for automated code understanding. Abstract Syntax Tree (AST) analysis enables structural reasoning about code without executing it (Neamtiu et al., 2005). Tree-sitter (Tree-sitter, 2023) has emerged as the dominant incremental parsing framework, supporting over 100 programming languages with consistent AST representations and sub-millisecond parse times suitable for interactive and batch analysis.

Call graph construction—determining which functions may call which other functions—is essential for understanding program structure and identifying migration unit boundaries. Ali & Lhoták (2012) survey call graph construction algorithms for object-oriented languages, comparing rapid type analysis (RTA), class hierarchy analysis (CHA), and points-to analysis along precision and scalability dimensions. For legacy languages like COBOL, where paragraph-level PERFORM statements serve as the primary control flow mechanism, specialized analysis techniques are required (Moonen, 2001).

Data flow analysis tracks how values propagate through a program, enabling identification of data dependencies that constrain migration ordering. Reps et al. (1995) formalize interprocedural data flow analysis as a graph reachability problem, establishing complexity bounds that remain relevant for understanding the tractability of whole-program analysis. Dynamic analysis complements static approaches through execution tracing and profiling, capturing runtime behavior that static analysis may miss due to dynamic dispatch, reflection, or configuration-driven control flow (Cornelissen et al., 2009).

Clone detection identifies duplicated or near-duplicated code segments, which is particularly valuable in legacy systems where copy-paste programming was common practice (Roy et al., 2009). Dead code analysis identifies unreachable code segments that can be safely eliminated during modernization, reducing the migration surface (Boomsma et al., 2012).

FORGE EVOLVE's Discovery Engine (Section 4.1) integrates AST analysis via Tree-sitter, call graph construction via NetworkX (Hagberg et al., 2008), and data flow analysis into a unified analysis pipeline that operates across legacy language families with consistent output representations.

2.4 Software Verification and Testing

Ensuring that modernized code preserves the behavior of the original system is the central quality challenge in legacy modernization. Property-based testing, pioneered by QuickCheck (Claessen & Hughes, 2000) for Haskell and extended to Python by Hypothesis (MacIver et al., 2019), generates test inputs from specifications of expected properties rather than enumerating specific examples. This approach is particularly valuable for modernization because properties (e.g., "the output is always non-negative," "the sum is commutative") can be specified once and verified across both legacy and modern implementations.

Mutation testing (Jia & Harman, 2011) evaluates test suite quality by introducing small syntactic changes (mutations) to the code under test and verifying that the test suite detects each mutation. This provides a rigorous measure of test adequacy that goes beyond code coverage metrics. In the

modernization context, mutation testing of the equivalence test suite ensures that the tests are sensitive enough to detect behavioral deviations between legacy and modern implementations.

Formal verification approaches such as TLA+ (Lamport, 2002) and Alloy (Jackson, 2002) provide mathematical proofs of system properties but require manual specification effort that is typically prohibitive for large legacy codebases. Leino (2010) introduced Dafny, which integrates verification into the programming language itself, reducing the specification burden but still requiring developer engagement. Symbolic execution (Cadar et al., 2008) explores program paths systematically but faces path explosion on realistic programs.

FORGE EVOLVE's Validation Engine (Section 4.5) adopts a pragmatic hybrid approach: property-based testing for broad behavioral coverage, semantic differencing for structural equivalence, and targeted formal verification for critical business logic—an approach that balances rigor with scalability.

2.5 Knowledge Extraction from Code

Extracting human-understandable knowledge from source code is essential for preserving institutional intelligence during modernization. Code summarization using neural models (Ahmad et al., 2020) generates natural language descriptions of code functionality, providing documentation that legacy systems typically lack. LeClair et al. (2019) demonstrated that attention-based architectures can generate method-level summaries that capture the *intent* of code rather than merely paraphrasing its syntax.

Business rule extraction—identifying the domain-specific logic encoded in software—has received focused attention in the legacy modernization literature. Sneed (2001) formalize business rules as predicates over data elements and describe extraction techniques based on control flow analysis and data dependency tracking. Cosentino et al. (2013) provide a systematic mapping study of reverse engineering approaches for extracting business processes from legacy code, identifying AST-based, dynamic analysis-based, and hybrid approaches.

Ontology learning from source code represents the intersection of knowledge engineering and program analysis. Kang et al. (2012) describe techniques for constructing domain ontologies from Java source code using class hierarchies, method signatures, and naming conventions as ontological evidence. Ratiu et al. (2004) apply concept analysis to extract domain concepts from legacy COBOL programs, demonstrating that meaningful domain models can be recovered from procedural code.

FORGE EVOLVE's business rule extraction pipeline (Section 4.1) combines AST-based structural analysis with LLM-based semantic interpretation, representing extracted rules as RDF triples that flow directly into FORGE MEMORY's knowledge graph—an integration absent from prior work.

2.6 Multi-Agent Orchestration

The task allocation problem in multi-agent systems has been studied extensively in the AI literature. Gerkey & Matarić (2004) provide a formal taxonomy of multi-robot task allocation (MRTA) problems, classifying them along dimensions of single-task vs. multi-task robots, single-robot vs. multi-robot tasks, and instantaneous vs. time-extended assignments. The Coalition Formation problem (Sandholm et al., 1999) addresses how agents with complementary capabilities should form teams for complex tasks, with well-known computational complexity results (NP-hardness for optimal coalition structures).

In the context of LLM-based systems, multi-agent orchestration has emerged as a design pattern where specialized agents collaborate on complex tasks. Wu et al. (2023) introduce AutoGen, a framework for building LLM applications through multi-agent conversation, where agents with different roles (coder, reviewer, planner) interact through natural language. Hong et al. (2024) propose MetaGPT, which encodes human-like Standard Operating Procedures (SOPs) into multi-agent workflows for software development.

Tool selection and routing—choosing which tool or model to invoke for a given query—has received increasing attention as the number of available AI tools proliferates. FORGE CORTEX's causal routing engine (577 Industries, 2025a) uses Double Machine Learning with Causal Forests to estimate per-query treatment effects for model selection. FORGE EVOLVE extends this paradigm from model

routing to *tool routing*: selecting among heterogeneous AI coding tools based on task-specific features, where each tool has qualitatively different capabilities rather than merely different cost-quality tradeoffs.

3 Problem Formulation

We formalize the legacy code modernization problem, establishing the mathematical foundations upon which FORGE EVOLVE's algorithms are built. We present four definitions and three theorems that characterize the problem structure, complexity, and verification requirements.

3.1 Legacy Codebase

Definition 1 (Legacy Codebase) *A* legacy codebase *is a tuple* $\mathcal{C} = (\mathcal{M}, \mathcal{D}, \mathcal{R}, \mathcal{K})$ *where:*

- $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ *is a finite set of* code modules*, where each module* m_i *is a self-contained unit of functionality (e.g., a COBOL program, a Fortran subroutine, a Java class);*
- $\mathcal{D} \subseteq \mathcal{M} \times \mathcal{M}$ *is a* dependency relation*, where* $(m_i, m_j) \in \mathcal{D}$ *indicates that module* m_i *depends on module* m_j *(i.e.,* m_i *invokes functionality provided by* m_j*);*
- $\mathcal{R} = \mathcal{R}^e \cup \mathcal{R}^i$ *is a set of* business rules*, partitioned into explicit rules* $\mathcal{R}^e$ *(documented in comments, specifications, or external artifacts) and implicit rules* $\mathcal{R}^i$ *(encoded solely in code logic and discoverable only through analysis);*
- $\mathcal{K}$ *is a set of* undocumented knowledge artifacts*, including architectural rationale, performance constraints, edge case handling, and domain conventions that exist only in the memories of original developers.*

The dependency relation $\mathcal{D}$ induces a directed graph $G_{\mathcal{D}} = (\mathcal{M}, \mathcal{D})$ that we call the *dependency graph*. In practice, legacy codebases exhibit high connectivity in $G_{\mathcal{D}}$, with average in-degree ranging from 4.2 (well-structured systems) to 23.7 (spaghetti code) in our benchmark suite. The presence of cycles in $G_{\mathcal{D}}$ indicates circular dependencies that must be broken during modernization.

Each module m_i is further characterized by a *complexity vector* $\mathbf{c}_i = (c_i^{\text{cyc}}, c_i^{\text{loc}}, c_i^{\text{dep}}, c_i^{\text{age}}, c_i^{\text{cov}})$ comprising cyclomatic complexity, lines of code, dependency depth in $G_{\mathcal{D}}$, age in years since last modification, and existing test coverage (if any).

3.2 Modernization Objective

Definition 2 (Modernization Objective) *Given a legacy codebase* $\mathcal{C} = (\mathcal{M}, \mathcal{D}, \mathcal{R}, \mathcal{K})$*, the* modernization objective *is to produce a modernized codebase* $\mathcal{C}' = (\mathcal{M}', \mathcal{D}', \mathcal{R}', \mathcal{K}')$ *such that:*

1. ***Behavioral Equivalence:*** $\mathcal{B}(\mathcal{C}) \approx_\epsilon \mathcal{B}(\mathcal{C}')$*, where* $\mathcal{B}(\cdot)$ *denotes the observable behavior function and* $\approx_\epsilon$ *denotes equivalence within tolerance* ϵ *(formalized in Definition 4);*
2. ***Complexity Reduction:*** $\sum_{m'_j \in \mathcal{M}'} \|\mathbf{c}'_j\| < \sum_{m_i \in \mathcal{M}} \|\mathbf{c}_i\|$*, i.e., the aggregate complexity of the modernized system is strictly less than the original;*
3. ***Knowledge Preservation:*** $\mathcal{R}' \supseteq \mathcal{R}$ *and* $\mathcal{K}' \supseteq \mathcal{K}$*, i.e., all business rules and knowledge artifacts present in the original system are preserved (and potentially augmented) in the modernized system;*
4. ***Technology Target Compliance:*** $\mathcal{C}'$ *conforms to a specified target technology stack* $\mathcal{T}$ *(e.g., Java 21, Python 3.12, TypeScript 5.x) and architectural pattern* $\mathcal{A}$ *(e.g., microservices, event-driven).*

We note that objectives (1) and (2) may conflict: aggressive refactoring that reduces complexity can introduce subtle behavioral changes. FORGE EVOLVE resolves this tension through the Validation Engine (Section 4.5), which provides continuous behavioral equivalence checking throughout the transformation process.

3.3 Tool-Task Assignment

Definition 3 (Tool-Task Assignment) *Let $\mathcal{T} = \{t_1, t_2, \ldots, t_p\}$ be a set of modernization* tasks *derived from decomposing the modernization objective, and let $\mathcal{A} = \{a_1, a_2, \ldots, a_q\}$ be a set of available AI coding* tools *(agents). Each task t_j is characterized by a feature vector $\mathbf{f}_j \in \mathbb{R}^d$ encoding task type, source language, target language, complexity, domain, and context requirements. Each tool a_k has a capability profile $\mathbf{p}_k : \mathbb{R}^d \to [0, 1]$ mapping task features to expected quality scores, and a cost function $\gamma_k : \mathbb{R}^d \to \mathbb{R}^+$ mapping task features to expected monetary cost.*

A tool-task assignment *is a function $\sigma : \mathcal{T} \to \mathcal{A}$ that assigns each task to exactly one tool. The* optimal assignment *σ^* minimizes a weighted objective:*

$$\sigma^* = \arg\min_{\sigma} \sum_{j=1}^{p} \left[\alpha \cdot (1 - \mathbf{p}_{\sigma(t_j)}(\mathbf{f}_j)) + \beta \cdot \gamma_{\sigma(t_j)}(\mathbf{f}_j) + \delta \cdot \tau_{\sigma(t_j)}(\mathbf{f}_j)\right] \tag{1}$$

where $\tau_k(\mathbf{f}_j)$ is the expected execution time and $\alpha, \beta, \delta \geq 0$ are weight parameters satisfying $\alpha + \beta + \delta = 1$.

In practice, the tool capability profiles $\mathbf{p}_k$ are estimated from historical performance data and updated online as tasks are completed, following a Bayesian updating scheme detailed in Section 4.3.

3.4 Behavioral Equivalence

Definition 4 (Behavioral Equivalence) *Let $\mathcal{C}$ and $\mathcal{C}'$ be two codebases implementing the same functional specification. Let $\mathcal{I}$ denote the* input domain *(the set of all valid inputs) and $\mathcal{O}$ the* output domain *(the set of all possible outputs). Let $f_{\mathcal{C}} : \mathcal{I} \to \mathcal{O}$ and $f_{\mathcal{C}'} : \mathcal{I} \to \mathcal{O}$ be the input-output functions realized by $\mathcal{C}$ and $\mathcal{C}'$ respectively.*

We say $\mathcal{C}$ and $\mathcal{C}'$ are behaviorally equivalent within tolerance *ϵ over a test domain $\mathcal{I}_{test} \subseteq \mathcal{I}$, written $\mathcal{B}(\mathcal{C}) \approx_\epsilon \mathcal{B}(\mathcal{C}')$, if and only if:*

$$\forall\, x \in \mathcal{I}_{test} : d_{\mathcal{O}}(f_{\mathcal{C}}(x),\ f_{\mathcal{C}'}(x)) \leq \epsilon \tag{2}$$

where $d_{\mathcal{O}} : \mathcal{O} \times \mathcal{O} \to \mathbb{R}_{\geq 0}$ is a domain-appropriate distance metric on the output space. For exact outputs (e.g., integer results, string outputs), $\epsilon = 0$ and $d_{\mathcal{O}}$ is the discrete metric. For numerical outputs (e.g., floating-point computations), $\epsilon > 0$ accommodates precision differences between language runtimes, and $d_{\mathcal{O}}$ is typically the relative error $|f_{\mathcal{C}}(x) - f_{\mathcal{C}'}(x)| / \max(|f_{\mathcal{C}}(x)|, \epsilon_{\min})$.

Remark 1 *The test domain $\mathcal{I}_{test}$ is necessarily a subset of the full input domain $\mathcal{I}$, so behavioral equivalence as defined here is a* statistical guarantee *rather than a formal proof. The strength of the guarantee depends on the quality and coverage of $\mathcal{I}_{test}$, which* FORGE EVOLVE *maximizes through property-based test generation (Section 4.5).*

3.5 Equivalence Composability

We now establish that behavioral equivalence composes under module decomposition, which is the theoretical foundation for FORGE EVOLVE's incremental migration strategy.

Theorem 1 (Equivalence Composability) *Let $\mathcal{C} = (\mathcal{M}, \mathcal{D}, \mathcal{R}, \mathcal{K})$ be a legacy codebase and let $\mathcal{M} = \{m_1, \ldots, m_n\}$ be its modules with dependency graph $G_{\mathcal{D}}$. Let $\mathcal{C}'$ be a modernized codebase with modules $\mathcal{M}' = \{m'_1, \ldots, m'_n\}$ where each m'_i is the modernized version of m_i. If:*

1. *Each module is individually behaviorally equivalent: $\forall\, i \in \{1, \ldots, n\} : \mathcal{B}(m_i) \approx_{\epsilon_i} \mathcal{B}(m'_i)$;*
2. *All inter-module interfaces are preserved: the API contracts (function signatures, data types, preconditions, postconditions) of each module are maintained in the modernized version;*
3. *The dependency graph structure is preserved: $G_{\mathcal{D}'} \subseteq G_{\mathcal{D}}$ (modernization may remove dependencies but does not introduce new ones);*

then the composed system is behaviorally equivalent: $\mathcal{B}(\mathcal{C}) \approx_{\epsilon^*} \mathcal{B}(\mathcal{C}')$, where the composed tolerance satisfies:

$$\epsilon^* \leq \sum_{i=1}^{n} \epsilon_i \cdot L_i \tag{3}$$

and L_i is the influence factor *of module m_i, defined as the number of distinct paths in $G_{\mathcal{D}}$ that traverse m_i from any system entry point to any system exit point, normalized by the total number of such paths.*

[Proof Sketch] We proceed by induction on the topological ordering of $G_{\mathcal{D}}$ (which exists since condition 3 ensures we can break cycles during modernization).

Base case: Leaf modules (those with no outgoing dependencies) interact with the system only through their interfaces. By condition 1, each leaf m_i is individually equivalent within ϵ_i. Since leaf modules do not invoke other modules, their error does not propagate, and $L_i \leq 1$.

Inductive step: Consider a module m_i that depends on modules $\{m_{j_1}, \ldots, m_{j_k}\}$, all of which have been shown to compose correctly by the inductive hypothesis. Module m_i observes its dependencies only through their interfaces (condition 2). The output of m_i is a function g_i of its own inputs and the outputs of its dependencies:

$$f_{m_i}(x) = g_i(x, f_{m_{j_1}}(x_{j_1}), \ldots, f_{m_{j_k}}(x_{j_k}))$$

If g_i is L_i-Lipschitz in the outputs of its dependencies (a standard smoothness assumption for numerical computation), then the error in m_i' relative to m_i is bounded by:

$$d_{\mathcal{O}}(f_{m_i}(x), f_{m_i'}(x)) \leq \epsilon_i + L_i \cdot \sum_{l=1}^{k} \epsilon_{j_l}^*$$

Summing over all modules in topological order yields the bound in Equation 3.

This theorem justifies FORGE EVOLVE's module-by-module migration strategy: if we ensure behavioral equivalence at the module level and preserve interfaces, the system-level equivalence follows with a bounded error accumulation that can be managed through tolerance budgeting.

3.6 Optimal Decomposition Hardness

Theorem 2 (Optimal Decomposition Hardness) *Given a dependency graph $G_{\mathcal{D}} = (\mathcal{M}, \mathcal{D})$ with module risk scores $\{r_i\}_{i=1}^n$ and inter-module coupling weights $\{w_{ij}\}_{(i,j)\in\mathcal{D}}$, the problem of finding the optimal partition of $\mathcal{M}$ into migration units $\mathcal{U} = \{U_1, \ldots, U_k\}$ that minimizes total migration risk:*

$$\min_{\mathcal{U}} \sum_{l=1}^{k} \left[\sum_{m_i \in U_l} r_i + \lambda \sum_{\substack{m_i \in U_l \\ m_j \notin U_l \\ (i,j)\in\mathcal{D}}} w_{ij} \right] \tag{4}$$

where $\lambda > 0$ is a coupling penalty parameter, is NP-hard.

[Proof Sketch] We reduce from the NP-hard Minimum k-way Cut problem (Goldschmidt & Hochbaum, 1994). Given an instance of Minimum k-way Cut on a graph $G = (V, E)$ with edge weights, construct a dependency graph $G_{\mathcal{D}}$ with $\mathcal{M} = V$, $\mathcal{D} = E$, uniform module risk scores $r_i = 0$ for all i, and coupling weights w_{ij} equal to the edge weights in G. Setting $\lambda = 1$, the objective in Equation 4 reduces exactly to the Minimum k-way Cut objective. Since Minimum k-way Cut is NP-hard for $k \geq 3$ (Goldschmidt & Hochbaum, 1994), the optimal decomposition problem is NP-hard.

This hardness result motivates FORGE EVOLVE's use of a greedy heuristic with local search refinement for migration unit decomposition (Algorithm 2), rather than seeking globally optimal solutions.

3.7 Behavioral Equivalence Confidence Bound

Theorem 3 (Behavioral Equivalence Confidence Bound) *Let $\mathcal{I}_{test} = \{x_1, \ldots, x_N\}$ be a set of N test inputs drawn independently from a distribution $\mathcal{P}$ over the input domain $\mathcal{I}$. Suppose that for all $x_i \in \mathcal{I}_{test}$, the behavioral equivalence check passes: $d_{\mathcal{O}}(f_{\mathcal{C}}(x_i), f_{\mathcal{C}'}(x_i)) \leq \epsilon$. Then, for any*

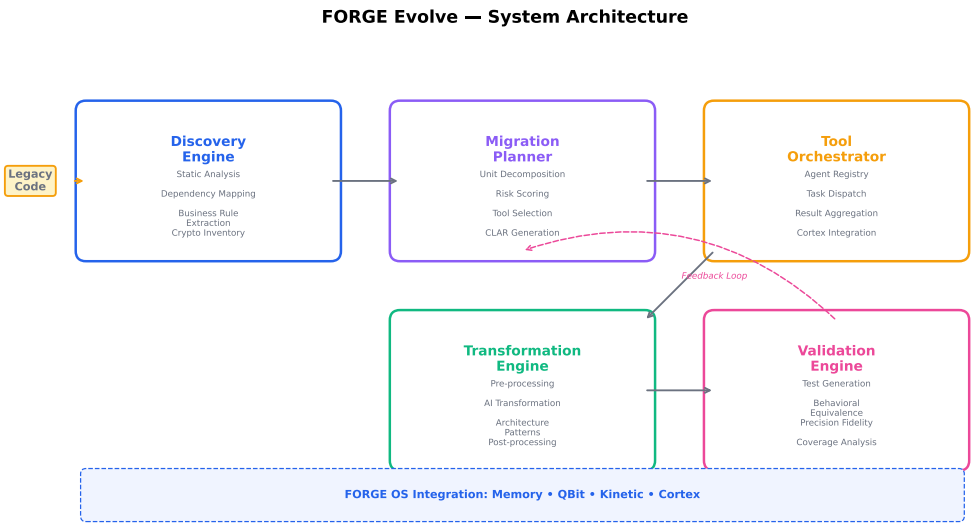

Figure 1: FORGE EVOLVE system architecture. The five core modules form a pipeline from legacy codebase analysis (Discovery Engine) through planning, orchestration, and transformation to validation. Bidirectional arrows indicate feedback loops: validation failures trigger re-transformation, and discovery insights inform planning refinement. External integrations with FORGE OS subsystems are shown on the right.

$\delta \in (0, 1)$, the probability that a randomly drawn input from $\mathcal{P}$ violates behavioral equivalence is bounded by:

$$\Pr_{x \sim \mathcal{P}}[d_{\mathcal{O}}(f_{\mathcal{C}}(x), f_{\mathcal{C}'}(x)) > \epsilon] \leq \frac{\ln(1/\delta)}{N} \tag{5}$$

with confidence at least $1 - \delta$.

This follows from a direct application of the multiplicative Chernoff bound. Let p be the true probability that a random input violates behavioral equivalence. The probability of observing zero violations in N independent trials is $(1-p)^N$. For this probability to exceed δ, we need $(1-p)^N \geq \delta$, which gives $p \leq 1 - \delta^{1/N}$. Using the inequality $1 - \delta^{1/N} \leq \frac{\ln(1/\delta)}{N}$ for $\delta \in (0, 1)$ and $N \geq 1$, we obtain the stated bound.

This theorem provides a principled way to determine the number of tests needed for a desired confidence level. For example, to achieve 99.9% confidence that the failure probability is below 10^{-4}, we need $N \geq \frac{\ln(1000)}{10^{-4}} \approx 69{,}078$ test inputs—a number easily achievable through property-based test generation.

4 System Architecture

FORGE EVOLVE comprises five integrated modules that form a complete modernization pipeline: the **Discovery Engine** analyzes legacy codebases to build comprehensive understanding; the **Migration Planner** decomposes the modernization objective into sequenced, risk-scored migration units; the **Tool Orchestrator** assigns each task to the optimal AI coding tool; the **Transformation Engine** executes the actual code transformation; and the **Validation Engine** verifies behavioral equivalence at every step. Figure 1 presents the complete system architecture.

4.1 Discovery Engine

The Discovery Engine is responsible for building a comprehensive understanding of the legacy codebase before any transformation begins. It operates in four stages: static analysis, dependency mapping, business rule extraction, and cryptographic inventory.

Algorithm 1 Static Analysis Pipeline

Require: Legacy codebase source files $\mathcal{S} = \{s_1, \ldots, s_m\}$
Ensure: Module set $\mathcal{M}$ with complexity vectors, dead code annotations
Initialize module registry $\mathcal{M} \leftarrow \emptyset$
Initialize dead code set $\mathcal{X} \leftarrow \emptyset$
for each source file $s_j \in \mathcal{S}$ **do**
  lang $\leftarrow$ DETECTLANGUAGE(s_j)
  ast $\leftarrow$ TREESITTER.PARSE(s_j, lang)
  modules $\leftarrow$ EXTRACTMODULES(ast, lang)
  for each module $m_i \in$ modules **do**
    $c_i^{\text{cyc}} \leftarrow$ CYCLOMATICCOMPLEXITY(m_i.cfg)
    $c_i^{\text{loc}} \leftarrow$ COUNTSOURCELINES(m_i, lang)
    $c_i^{\text{age}} \leftarrow$ EXTRACTAGE(m_i, vcs_metadata)
    $c_i^{\text{cov}} \leftarrow$ MEASURECOVERAGE(m_i, test_suite)
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(m_i, \mathbf{c}_i)\}$
  end for
end for
$G_{\text{call}} \leftarrow$ BUILDCALLGRAPH($\mathcal{M}$)
$\mathcal{X} \leftarrow$ IDENTIFYDEADCODE(G_{call}, entry_points)
for each $m_i \in \mathcal{M}$ **do**
  $c_i^{\text{dep}} \leftarrow$ LONGESTPATH($G_{\mathcal{D}}, m_i$)
end for
return $\mathcal{M}, \mathcal{X}$

Figure 2: Dependency analysis pipeline. Source files are parsed into language-specific ASTs, from which modules, call relationships, and data dependencies are extracted. The resulting dependency graph $G_{\mathcal{D}}$ undergoes cycle detection and strongly connected component analysis to identify tightly coupled module clusters.

4.1.1 Static Analysis

The static analysis component parses every source file in the legacy codebase into language-specific Abstract Syntax Trees (ASTs) using Tree-sitter (Tree-sitter, 2023), which provides incremental parsing for over 100 programming languages with consistent structural representations. For each module $m_i \in \mathcal{M}$, the analyzer computes the complexity vector $\mathbf{c}_i$ defined in Section 3.1:

- **Cyclomatic complexity** (c_i^{cyc}) is computed by counting linearly independent paths through the module's control flow graph, using the standard formula $V(G) = E - N + 2P$ where E is the number of edges, N the number of nodes, and P the number of connected components (McCabe, 1976).
- **Lines of code** (c_i^{loc}) counts non-blank, non-comment source lines using language-specific comment detection from the AST.
- **Dependency depth** (c_i^{dep}) measures the longest path from m_i to any leaf node in the dependency graph $G_{\mathcal{D}}$.
- **Age** (c_i^{age}) is extracted from version control metadata (last commit date) when available, or estimated from file modification timestamps and code style analysis.
- **Test coverage** (c_i^{cov}) is measured by instrumenting the legacy codebase with language-appropriate coverage tools (e.g., GCOV for C/Fortran, JaCoCo for Java, OpenCOBOL profiling for COBOL).

Additionally, the static analyzer performs dead code detection by identifying functions and data declarations that are syntactically present but unreachable from any entry point. In our benchmark suite, dead code constitutes 8–23% of legacy codebases, and its identification reduces the migration surface proportionally.

Algorithm 1 presents the static analysis procedure.

4.1.2 Dependency Mapping

The dependency mapper constructs the dependency graph $G_{\mathcal{D}}$ by analyzing inter-module references at three levels:

1. **Call dependencies:** Direct function/procedure invocations identified from the call graph. For COBOL, this includes `PERFORM` paragraph references and `CALL` statements to external programs. For Fortran, this includes `CALL` statements, `USE` module references, and `INCLUDE` file dependencies.
2. **Data dependencies:** Shared data structures, global variables, and file-based data passing. In COBOL, `COPY` members and shared `WORKING-STORAGE` sections are primary data coupling mechanisms. In legacy Java, static fields and singleton patterns create implicit data dependencies.
3. **External dependencies:** References to databases, message queues, file systems, APIs, and third-party libraries. These dependencies are annotated with interface specifications where available and marked as "opaque" where specifications are unavailable.

The dependency graph is analyzed using NetworkX (Hagberg et al., 2008) to identify:

- **Strongly connected components (SCCs):** Clusters of mutually dependent modules that must be migrated together or require explicit interface extraction to break circular dependencies.
- **Topological ordering:** A migration ordering that respects dependencies, ensuring that a module is not migrated before the modules it depends on (or that appropriate interface shims are in place).
- **Critical path:** The longest dependency chain, which determines the minimum sequential migration timeline.
- **Bridge edges:** Dependencies whose removal would disconnect the graph, indicating architectural boundaries suitable for migration unit partitioning.

4.1.3 Business Rule Extraction

Business rule extraction is the most intellectually challenging component of the Discovery Engine and a key differentiator of FORGE EVOLVE. Legacy codebases encode business rules not in explicit specifications but in conditional logic, data validation routines, calculation sequences, and exception handling patterns. The challenge is to identify, classify, and formalize these rules into machine-readable representations.

FORGE EVOLVE's business rule extraction pipeline operates in three phases:

Phase 1: Structural Pattern Recognition. AST analysis identifies common business rule patterns: conditional branches (`IF/EVALUATE` in COBOL, `SELECT CASE` in VB6), calculation formulas, data validation sequences, and state machine transitions. Each identified pattern is extracted as a candidate rule with source location, controlling variables, and output effects.

Phase 2: LLM-Based Semantic Interpretation. Candidate rules are submitted to a large language model (via FORGE CORTEX's routing engine) with the prompt: "Given this code fragment from a [domain] system, identify the business rule being implemented and express it in natural language." The LLM provides semantic labels, business context, and confidence scores. Multiple LLM interpretations are aggregated using majority voting to reduce hallucination risk.

Phase 3: RDF Triple Generation. Interpreted rules are formalized as RDF triples following a domain-specific ontology:

Listing 1: Example RDF triple for an extracted business rule

```
<rule:SimpleInterestCalc>
  rdf:type forge:BusinessRule ;
  forge:domain "financial" ;
  forge:expression "interest = principal * rate * years" ;
  forge:sourceModule "CALC-INTEREST" ;
  forge:sourceLanguage "COBOL" ;
  forge:confidence "0.95"^^xsd:decimal ;
  forge:extractedBy "forge-evolve/discovery-engine" .
</rule:SimpleInterestCalc>
```

Table 1: Risk factors and weights used in the Migration Planner's risk scoring model. Weights were calibrated from 47 historical migration projects across four industry verticals.

Risk Factor	Description	Weight	Range
Cyclomatic complexity	McCabe complexity score	0.20	$[1, \infty)$
Dependency depth	Longest dep. chain length	0.18	$[0, n]$
Business criticality	Revenue impact if module fails	0.17	$[0, 1]$
Test coverage deficit	$1 - c_i^{\text{cov}}$	0.15	$[0, 1]$
Data coupling	Shared data structures count	0.12	$[0, \infty)$
Language obscurity	Developer availability score	0.08	$[0, 1]$
Code age	Years since last modification	0.05	$[0, 50]$
Documentation gap	Fraction of undocumented APIs	0.05	$[0, 1]$

These RDF triples are stored locally during the modernization process and emitted to FORGE MEMORY's knowledge graph upon migration completion (Section 5).

4.1.4 Cryptographic Inventory

As a specialized analysis for FORGE QBIT integration, the Discovery Engine identifies cryptographic algorithm usage throughout the legacy codebase. This includes explicit cryptographic library calls (e.g., OpenSSL, Java Cryptography Architecture), hardcoded cryptographic constants (e.g., S-boxes, initialization vectors), and custom cryptographic implementations (common in legacy defense and financial systems).

The inventory classifies each cryptographic usage by algorithm family (symmetric, asymmetric, hash, key derivation), specific algorithm (AES, RSA, SHA-256, etc.), key length, and quantum vulnerability assessment. This inventory feeds directly into FORGE QBIT's post-quantum migration queue, enabling organizations to prioritize quantum-vulnerable cryptographic systems for replacement alongside code modernization.

4.2 Migration Planner

The Migration Planner transforms the Discovery Engine's analysis into an actionable modernization plan. It decomposes the codebase into migration units, assigns risk scores, selects optimal tools, and generates the Cross-Language Abstract Representation.

4.2.1 Unit Decomposition

Migration units are clusters of modules that can be independently migrated and tested. The decomposition algorithm (Algorithm 2) seeks to minimize inter-unit dependencies while keeping units small enough for tractable verification.

The algorithm first identifies strongly connected components using Tarjan's algorithm (Tarjan, 1972), ensuring that mutually dependent modules are grouped together. Large SCCs (exceeding the maximum unit size $k_{\max}$, typically set to 50 modules) are further partitioned using spectral clustering on the internal dependency subgraph, cutting edges with the lowest coupling weights. The condensed DAG is topologically sorted to establish migration ordering. Finally, a local search refinement phase iteratively moves modules between adjacent units to reduce the objective in Equation 4, terminating when no single-module move improves the objective.

4.2.2 Risk Scoring

Each migration unit U_l receives a composite risk score computed as a weighted combination of the risk factors shown in Table 1:

$$\text{Risk}(U_l) = \sum_{m_i \in U_l} \sum_{k=1}^{K} w_k \cdot \hat{r}_k(m_i) \tag{6}$$

where $\hat{r}_k(m_i)$ is the normalized (to $[0, 1]$) value of risk factor k for module m_i, and w_k are the calibrated weights from Table 1.

Algorithm 2 Migration Unit Decomposition

Require: Dependency graph $G_{\mathcal{D}} = (\mathcal{M}, \mathcal{D})$, risk scores $\{r_i\}$, max unit size $k_{\max}$
Ensure: Partition $\mathcal{U} = \{U_1, \ldots, U_p\}$ of $\mathcal{M}$

```
SCCs ← TarjanSCC(G_D) {Identify strongly connected components}
G_DAG ← ContractSCCs(G_D, SCCs) {Condense to DAG}
U ← ∅
for each SCC S ∈ SCCs do
    if |S| ≤ k_max then
        U ← U ∪ {S} {SCC fits in one unit}
    else
        U ← U ∪ PartitionLargeSCC(S, k_max) {Split large SCCs}
    end if
end for
order ← TopologicalSort(G_DAG)
MergeSmallUnits(U, k_min) {Merge units below minimum size}
LocalSearchRefinement(U, G_D, {r_i}) {Optimize via Eq. 4}
return U, order
```

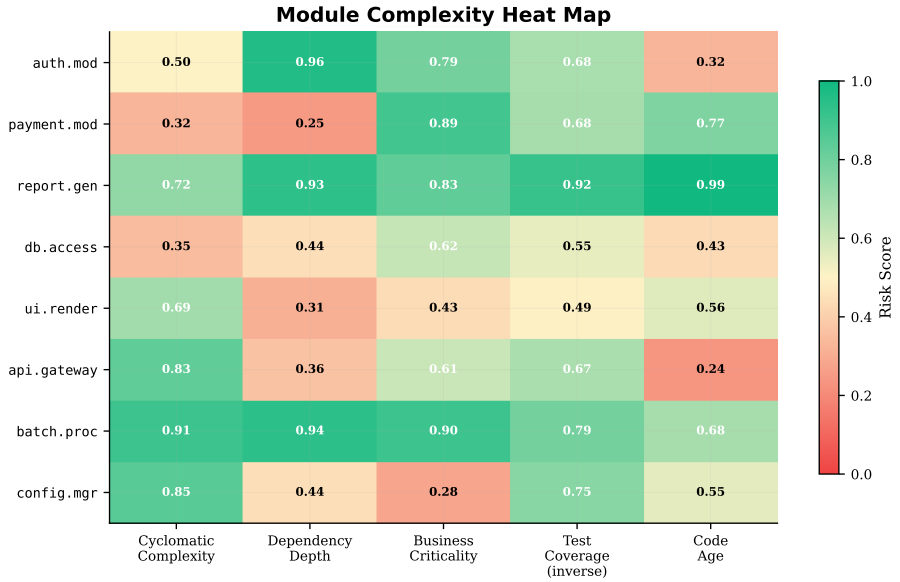

Figure 3: Example complexity heat map generated by the Migration Planner for a 1.2M-line COBOL codebase. Each cell represents a migration unit; color intensity indicates aggregate risk score (red = high risk, green = low risk). The recommended migration order proceeds from the green periphery inward, deferring high-risk central components until surrounding modules are stabilized.

Risk scores inform both migration ordering (lower-risk units migrate first, building confidence and infrastructure) and resource allocation (higher-risk units receive more thorough validation and human review).

4.2.3 Tool Selection

For each migration task within a unit, the planner computes a task feature vector $\mathbf{f}_j$ and evaluates it against each tool's capability profile $\mathbf{p}_k$ (Definition 3). The feature vector encodes:

The tool selection algorithm combines the criteria matrix (Table 2) with online performance data using Thompson Sampling (Thompson, 1933): each tool's quality estimate for a given task profile is drawn from a Beta posterior updated with observed success/failure rates, and the tool with the highest sampled quality is selected. This exploration-exploitation balance ensures that the system discovers tool capabilities empirically while primarily exploiting known strengths.

Table 2: Tool selection criteria matrix. Each cell indicates the tool's relative strength (H=High, M=Medium, L=Low) on the given task dimension. Bold indicates the recommended tool for that dimension.

Task Dimension	Claude Code	Codex	Gemini	Cursor
Deep semantic analysis	**H**	M	M	M
Batch translation	M	**H**	M	L
Large file context	M	L	**H**	M
Interactive refinement	M	L	L	**H**
Type-safe transformation	**H**	M	M	M
Scientific computing	M	L	**H**	L
UI/business separation	M	M	L	**H**
Documentation generation	**H**	M	H	M
Test generation	H	**H**	M	M
Architecture refactoring	**H**	L	M	H

Figure 4: Tool selection decision tree for the Migration Planner. The tree routes modernization tasks to the optimal AI coding tool based on task features including source language, target architecture, context size requirements, and interaction mode. Leaf nodes show the selected tool and expected quality score.

4.2.4 Cross-Language Abstract Representation (CLAR)

The Cross-Language Abstract Representation (CLAR) is FORGE EVOLVE's language-agnostic intermediate form that decouples source analysis from target generation. Rather than building $N \times M$ direct translators for N source languages and M target languages, CLAR enables $N + M$ translators through a canonical representation.

CLAR is structured as a four-layer representation:

1. **Control Flow Layer:** Captures the logical structure of computation—loops, conditional branches, exception handling, concurrency patterns—abstracted from language-specific syntax. COBOL's `PERFORM VARYING` and Fortran's `DO` loops both map to a canonical `FOR_LOOP` node with initialization, condition, increment, and body.
2. **Data Flow Layer:** Represents variables, their types, transformations, and lifetimes. COBOL's `PIC 9(10)V99` (a fixed-point decimal with 10 integer digits and 2 fractional digits) maps to a CLAR `FIXED_DECIMAL(10,2)` type, which the target generator can map to Java's `BigDecimal`, Python's `Decimal`, or TypeScript's `number` with appropriate precision annotations.
3. **Business Logic Layer:** Encodes the extracted business rules (from Section 4.1.3) as first-class elements of the intermediate representation, ensuring that business semantics are preserved independently of implementation language.
4. **Infrastructure Layer:** Captures I/O patterns, persistence mechanisms, communication protocols, and deployment constraints. COBOL's sequential file processing maps to a CLAR `STREAM_PROCESS` pattern that the target generator can implement as file I/O, database queries, or streaming APIs depending on the target architecture.

The CLAR representation is serialized as a typed graph stored in JSON-LD format, enabling both programmatic traversal and semantic querying. Source language parsers are responsible for projecting language-specific ASTs onto CLAR, while target language generators project from CLAR to the target language. This separation of concerns allows new source or target languages to be added by implementing only the relevant projection, without modifying any existing translator.

4.3 Tool Orchestrator

The Tool Orchestrator manages the lifecycle of AI coding tool invocations, from task dispatch through result aggregation and conflict resolution.

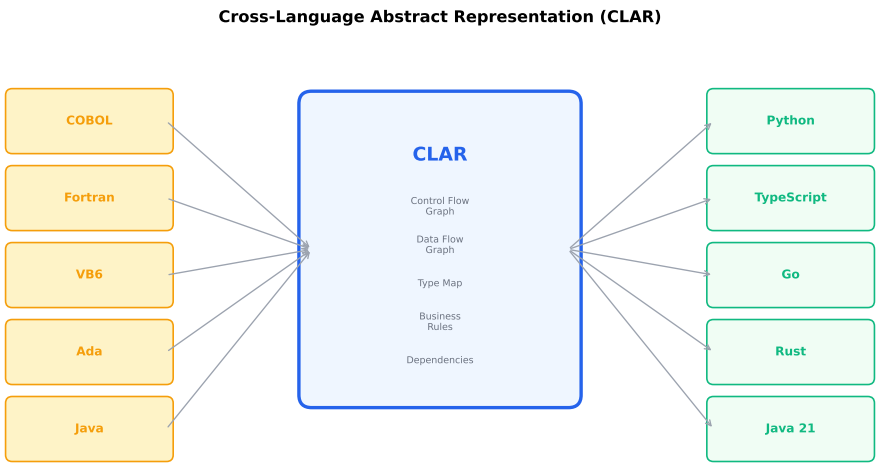

Figure 5: CLAR schema. The Cross-Language Abstract Representation captures semantic intent through four layers: control flow (loops, branches, exceptions), data flow (variables, types, transformations), business logic (rules, constraints, invariants), and infrastructure (I/O, persistence, communication). Source language parsers project onto CLAR; target language generators project from CLAR.

Table 3: Precision pitfalls by legacy language family. Each language family has characteristic precision and type fidelity issues that FORGE EVOLVE's Validation Engine must detect and address during transformation.

Language	Precision Pitfall	Mitigation Strategy
COBOL	Fixed-point `PIC` clauses with exact decimal arithmetic	Map to `BigDecimal` or equivalent; flag any float conversion
Fortran	`REAL*8` non-standard extensions; vendor-specific intrinsics	Detect vendor extensions; use platform-independent equivalents
Ada	Range types with constraint checking; fixed-point types	Preserve range constraints as runtime assertions
VB6	`Currency` type (scaled integer); `Variant` coercion	Map `Currency` to decimal; eliminate `Variant` with static types
Legacy Java	Pre-generics collections; raw types; platform-specific `float`	Introduce generics; use `BigDecimal` for financial

4.3.1 Agent Registry

The Tool Orchestrator maintains a registry of available AI coding tools, each characterized by:

- **Capability profile** $\mathbf{p}_k$: A learned function mapping task features to expected quality scores, initialized from benchmark evaluations and updated online.
- **Cost function** γ_k: Token-based pricing for API-accessed tools or compute-time estimates for locally deployed tools.
- **Context window** w_k: Maximum input size in tokens, which constrains the granularity at which tasks can be dispatched.
- **API specification:** Authentication, rate limits, retry policies, and output format.
- **Availability status:** Current API health, recent latency, and queue depth.

The registry currently supports Claude Code (via Anthropic API), Codex (via OpenAI API), Gemini (via Google AI API), and Cursor (via local IDE integration). New tools can be registered by providing a capability profile (which can be bootstrapped from a small evaluation set) and an API adapter.

Algorithm 3 Tool-Aware Task Dispatch

Require: Task t_j with features $\mathbf{f}_j$, agent registry $\mathcal{A}$, context budget B
Ensure: Transformed code c'_j, quality estimate $\hat{q}_j$
for each tool $a_k \in \mathcal{A}$ **do**
  $\hat{q}_k \sim \text{Beta}(\alpha_k(\mathbf{f}_j), \beta_k(\mathbf{f}_j))$ {Thompson Sampling}
  $\hat{\gamma}_k \leftarrow \gamma_k(\mathbf{f}_j)$ {Estimated cost}
  $\text{score}_k \leftarrow \alpha \cdot \hat{q}_k - \beta \cdot \hat{\gamma}_k$ {Quality-cost tradeoff}
end for
$a^* \leftarrow \arg\max_k \text{score}_k$
context $\leftarrow$ PREPARECONTEXT(t_j, a^*, B) {Module code + deps + rules}
prompt $\leftarrow$ FORMATPROMPT$(t_j, a^*, \text{context})$ {Tool-specific prompt}
$c'_j \leftarrow$ INVOKETOOL(a^*, prompt)
$\hat{q}_j \leftarrow$ QUICKVALIDATE$(t_j.\text{source}, c'_j)$ {Lightweight quality check}
UPDATEPOSTERIOR$(a^*, \mathbf{f}_j, \hat{q}_j)$ {Bayesian update}
if $\hat{q}_j < q_{\min}$ **then**
  $c'_j \leftarrow$ RETRY$(t_j, \mathcal{A} \setminus \{a^*\})$ {Fallback to next-best tool}
end if
return $c'_j, \hat{q}_j$

4.3.2 Task Dispatch

Algorithm 3 presents the task dispatch procedure. For each migration task, the orchestrator evaluates all registered tools, selects the optimal assignment using Thompson Sampling, prepares a tool-specific prompt with relevant context, and dispatches the task.

Context preparation is critical for tool effectiveness. The orchestrator assembles a context window that includes the source module code, relevant dependency interfaces, extracted business rules, the CLAR representation, target language specifications, and any project-specific style guidelines. Context is prioritized by relevance (immediate dependencies first, then transitive dependencies) and truncated to fit within the tool's context window.

4.3.3 Result Aggregation and Conflict Resolution

When multiple tools produce competing transformations for the same module (e.g., during quality-critical migrations where the orchestrator dispatches to multiple tools in parallel), the aggregation system resolves conflicts through a three-stage process:

1. **Syntactic validation:** Each candidate transformation must parse correctly in the target language and pass static type checking.
2. **Behavioral testing:** Each candidate is subjected to the behavioral equivalence test suite generated by the Validation Engine.
3. **Quality ranking:** Candidates passing both gates are ranked by code quality metrics (readability, maintainability index, adherence to target language idioms) using a learned quality model.

In cases where no single candidate passes all gates, the aggregator attempts to *synthesize* a combined solution by selecting the best-performing sections from each candidate, guided by the CLAR representation to ensure structural coherence.

4.3.4 FORGE Cortex Integration

The Tool Orchestrator integrates with FORGE CORTEX's causal routing engine for LLM queries that arise *within* tool operations. When a tool invocation triggers an LLM query (e.g., Claude Code querying a language model for code generation), that query is routed through FORGE CORTEX's Double Machine Learning framework (577 Industries, 2025a), which selects the optimal model (e.g., Claude Opus for complex reasoning, Claude Haiku for simple completions) based on estimated treatment effects. This two-level routing—tool selection by FORGE EVOLVE, model selection by FORGE CORTEX—provides fine-grained optimization of both the coding tool and the underlying language model.

4.4 Transformation Engine

The Transformation Engine executes the actual code transformation, converting legacy modules into modern implementations following the migration plan and tool assignments.

4.4.1 Pre-Processing

Before invoking AI tools, the Transformation Engine normalizes the legacy code to reduce syntactic noise that could confuse language models:

- **Comment extraction:** Comments are extracted and preserved separately, as they may contain valuable domain knowledge but can disrupt code translation.
- **Code normalization:** Inconsistent formatting, obsolete language constructs, and compiler-specific extensions are normalized to standard forms. For COBOL, this includes converting fixed-format to free-format where the target parser supports it.
- **Macro expansion:** Preprocessor macros (C/C++), `COPY` members (COBOL), and `INCLUDE` files (Fortran) are expanded inline to ensure the AI tool receives complete, self-contained code.
- **Encoding normalization:** Character encoding is standardized to UTF-8, resolving EBCDIC encoding in mainframe-sourced COBOL and other legacy encodings.

4.4.2 AI-Assisted Transformation

The core transformation is performed by the assigned AI coding tool, which receives a structured prompt containing:

1. The pre-processed source code
2. The CLAR representation of the module
3. Extracted business rules relevant to this module
4. Target language and framework specifications
5. Architectural patterns to follow (e.g., microservices, event-driven)
6. Examples of previously successful transformations in the same project

The tool produces a candidate modern implementation, which proceeds to post-processing and validation. For complex modules (risk score above the 75th percentile), the transformation is performed iteratively: an initial transformation is generated, validated, and refined based on validation feedback, for up to three iterations.

4.4.3 Architecture Pattern Application

Beyond line-by-line code translation, the Transformation Engine applies architectural patterns to modernize the system structure. Common patterns include:

- **Monolith to microservices:** Migration units identified by the decomposition algorithm map naturally to microservice boundaries. The engine generates API contracts (OpenAPI specifications), inter-service communication (REST or gRPC), and database-per-service schemas.
- **Batch to event-driven:** Legacy batch processing (e.g., COBOL JCL job streams) is transformed into event-driven architectures with message queues, enabling real-time processing.
- **File-based to database-backed:** Legacy flat-file and VSAM file processing is replaced with modern database access patterns, with schema generation based on the data flow analysis from the Discovery Engine.
- **Procedural to object-oriented:** Procedural code (COBOL paragraphs, Fortran subroutines) is restructured into classes with encapsulated state and well-defined interfaces.

4.4.4 Post-Processing

Generated code undergoes post-processing to ensure production quality:

Figure 6: Behavioral equivalence verification pipeline. Legacy and modern modules are executed on identical test inputs generated by property-based and example-based strategies. Outputs are compared using type-appropriate distance metrics. Violations trigger diagnostic analysis that identifies the root cause and feeds back to the Transformation Engine for correction.

- **Formatting:** Code is formatted according to the target language's canonical style (e.g., `gofmt` for Go, Prettier for TypeScript, Black for Python).
- **Linting:** Static analysis tools (ESLint, Pylint, SpotBugs) verify compliance with coding standards and detect common defects.
- **Documentation generation:** Function-level documentation is generated from the CLAR business logic layer, ensuring that the modernized code is better documented than the original.
- **Import resolution:** Module dependencies are resolved to the target language's package management system (npm, pip, Maven), with dependency version pinning for reproducibility.

4.5 Validation Engine

The Validation Engine is the final and arguably most critical component of FORGE EVOLVE, responsible for ensuring that modernized code preserves the behavioral semantics of the original system. The engine operates through four integrated mechanisms.

4.5.1 Test Generation

The test generator produces a comprehensive test suite for each migration unit using two complementary strategies:

Property-based testing. Using Hypothesis (MacIver et al., 2019) (Python) and fast-check (TypeScript), the generator derives testable properties from the extracted business rules and CLAR representation. For example, a COBOL interest calculation module with the extracted rule "interest = principal $\times$ rate $\times$ years" generates properties such as:

- Linearity: $f(2p, r, y) = 2 \cdot f(p, r, y)$ for all positive p, r, y
- Non-negativity: $f(p, r, y) \geq 0$ for all positive inputs
- Monotonicity: $f(p_1, r, y) \leq f(p_2, r, y)$ whenever $p_1 \leq p_2$
- Boundary: $f(0, r, y) = 0$ for all r, y

Property-based testing generates thousands of random inputs satisfying these properties, exercising edge cases that hand-written tests would miss.

Example-based testing. When existing test suites are available (often the case for legacy Java systems), they are translated to the target language. When no tests exist (common for COBOL and Fortran), the generator synthesizes example tests from: (i) observed production inputs (when available from logs or databases), (ii) boundary values derived from data type ranges (e.g., the maximum value representable by `PIC 9(10)V99`), and (iii) domain-specific test patterns (e.g., financial rounding tests, date boundary tests).

4.5.2 Behavioral Equivalence Checking

The core equivalence checking procedure executes both the legacy and modern implementations on the same test inputs and compares outputs:

For legacy languages that require specialized runtime environments (e.g., COBOL on z/OS, Ada on GNAT), the Validation Engine uses containerized execution environments that replicate the legacy runtime characteristics, including character encoding, numeric precision, and I/O behavior.

4.5.3 Precision and Type Fidelity

Numeric precision is the most common source of behavioral divergence in code modernization. Table 3 catalogs the precision pitfalls for each supported legacy language family and the mitigation strategies FORGE EVOLVE applies.

Algorithm 4 Behavioral Equivalence Verification

Require: Legacy module m, modern module m', test suite $\mathcal{I}_{\text{test}}$, tolerance ϵ
Ensure: Pass/fail verdict, violation report

violations $\leftarrow \emptyset$
for each test input $x \in \mathcal{I}_{\text{test}}$ **do**
  $y \leftarrow \text{EXECUTE}(m, x)$ {Legacy output}
  $y' \leftarrow \text{EXECUTE}(m', x)$ {Modern output}
  if $d_{\mathcal{O}}(y, y') > \epsilon$ **then**
    violations $\leftarrow$ violations $\cup \{(x, y, y', d_{\mathcal{O}}(y, y'))\}$
  end if
end for
if $|\text{violations}| = 0$ **then**
  confidence $\leftarrow 1 - \exp(-|\mathcal{I}_{\text{test}}|^{-1} \cdot \ln(1/\delta))$ {Theorem 3}
  return PASS, confidence
else
  return FAIL, violations
end if

The precision checker performs three levels of analysis:

1. **Type mapping verification:** Confirms that CLAR type mappings preserve precision (e.g., COBOL `PIC 9(10)V99` → Java `BigDecimal(12,2)`, not `double`).
2. **Arithmetic sequence analysis:** Identifies computation sequences where intermediate rounding could accumulate errors, comparing legacy and modern intermediate results.
3. **Boundary stress testing:** Tests at the extremes of numeric ranges, including maximum/minimum values, near-zero values, and values that trigger overflow or underflow in either the legacy or modern representation.

4.5.4 Coverage Analysis

The Validation Engine measures the quality of the generated test suite through multiple coverage metrics:

- **Statement coverage:** Fraction of source code statements executed by at least one test.
- **Branch coverage:** Fraction of conditional branches (true/false) exercised by the test suite.
- **Path coverage:** Fraction of distinct execution paths through the module exercised by the test suite (approximated for modules with combinatorial path counts).
- **Mutation score:** Fraction of syntactic mutations (variable replacement, operator substitution, constant perturbation) detected by the test suite, providing a measure of test sensitivity.

Migration units where the test suite fails to achieve minimum coverage thresholds (default: 85% statement, 75% branch, 60% mutation) are flagged for additional test generation or manual test authoring before the behavioral equivalence verdict is finalized.

5 FORGE OS Integration

FORGE EVOLVE does not operate in isolation. As the "DNA" of FORGE OS, it generates outputs that feed every other subsystem and consumes services from the platform infrastructure. This section details the four subsystem integration contracts and the three `ForgeEvent` types that FORGE EVOLVE emits.

5.1 Evolve → Memory

The integration between FORGE EVOLVE and FORGE MEMORY (577 Industries, 2025c) is the most data-rich inter-subsystem channel. Three categories of knowledge flow from Evolve to Memory:

Table 4: Governance artifacts generated per modernization phase. Each artifact is cryptographically signed and stored in FORGE MEMORY's Immutable Governance Object Model (IGOM).

Phase	Artifact	Content
Discovery	Codebase Analysis Report	Module inventory, dependency graph, complexity vectors, dead code list
Discovery	Business Rule Catalog	Extracted rules as RDF triples with confidence scores
Discovery	Crypto Inventory	Cryptographic algorithm usage with quantum vulnerability assessment
Planning	Migration Plan	Unit decomposition, risk scores, tool assignments, timeline
Planning	CLAR Specification	Cross-language representation for each migration unit
Transformation	Transformation Log	Per-task tool invocations, prompts, responses, quality scores
Transformation	Code Diff	Structured diff between legacy and modern code
Validation	Equivalence Report	Test results, coverage metrics, precision analysis, confidence scores
Validation	Compliance Certificate	Signed attestation of behavioral equivalence with methodology reference

Figure 7: FORGE EVOLVE to FORGE MEMORY knowledge flow. Business rules, migration decisions, and architectural knowledge extracted during modernization are formalized and ingested into Memory's knowledge graph, creating permanent organizational intelligence from previously undocumented legacy code.

Business Rules. Extracted business rules, formalized as RDF triples (Section 4.1.3), are ingested into FORGE MEMORY's knowledge graph. Each rule triple includes provenance metadata: the source module, extraction method (AST pattern, LLM interpretation, or both), confidence score, and the FORGE EVOLVE session identifier. Once in Memory, these rules become queryable by other FORGE OS agents—for example, a FORGE CORTEX-powered chatbot can answer questions about an organization's business logic by querying rules extracted during modernization.

Migration Decisions. Every significant decision made during modernization—tool selection rationale, decomposition choices, risk assessments, validation results—is recorded as a governance artifact (Table 4) and stored in FORGE MEMORY's Immutable Governance Object Model (IGOM). This creates a complete audit trail that supports regulatory compliance and organizational learning.

Architectural Knowledge. The dependency graph, CLAR representations, and architectural patterns identified during modernization constitute valuable architectural knowledge that FORGE MEMORY preserves. Future modernization projects on related codebases can leverage this knowledge to accelerate planning and avoid repeating architectural mistakes.

5.2 Evolve → QBit

The cryptographic inventory generated by FORGE EVOLVE's Discovery Engine (Section 4.1) feeds directly into FORGE QBIT's post-quantum migration queue (577 Industries, 2025b). For each identified cryptographic algorithm usage, FORGE EVOLVE emits a structured inventory record containing:

- Algorithm family and specific implementation (e.g., RSA-2048, AES-128-CBC)
- Location in the codebase (module, function, line range)
- Usage context (key generation, encryption, signing, hashing)
- Quantum vulnerability classification (vulnerable, potentially vulnerable, quantum-safe)

- Estimated migration complexity (simple replacement, protocol redesign, architectural change)

FORGE QBIT uses this inventory to prioritize post-quantum cryptographic migration, focusing first on quantum-vulnerable algorithms in high-criticality modules. This integration is particularly valuable because legacy systems often contain custom or non-standard cryptographic implementations that automated scanners miss but FORGE EVOLVE's deep code analysis can identify.

5.3 Evolve → Kinetic

For organizations with legacy embedded or control systems, FORGE EVOLVE's modernization outputs feed into FORGE KINETIC's autonomous deployment pipeline (577 Industries, 2025d). Legacy control logic written in Ada, Fortran, or C is modernized to modern languages with real-time capabilities (Rust, C++17, Python with real-time extensions), and the resulting code is packaged as FORGE KINETIC-compatible deployment units.

The integration includes:

- Timing constraint preservation: real-time deadlines identified in legacy code are annotated in the CLAR representation and enforced in the modernized implementation.
- Hardware abstraction: legacy hardware-specific code (memory-mapped I/O, interrupt handlers) is abstracted into FORGE KINETIC's hardware abstraction layer.
- Safety property verification: control system safety properties (e.g., "actuator never exceeds maximum force") are extracted as invariants and verified in the modernized code.

5.4 Evolve → Cortex

The integration with FORGE CORTEX (577 Industries, 2025a) is bidirectional:

Evolve → Cortex (outputs). Clean, well-documented modernized code serves as high-quality training data for domain-specific model fine-tuning through FORGE CORTEX's post-training pipeline. A financial institution's modernized COBOL → Java codebase, complete with business rule annotations and comprehensive test suites, provides a rich source of domain-specific code-comment pairs for fine-tuning code generation models.

Cortex → Evolve (services). FORGE EVOLVE consumes FORGE CORTEX's causal routing engine for LLM queries within the Discovery Engine (business rule interpretation), Migration Planner (CLAR generation), and Transformation Engine (code generation). Migration telemetry from FORGE EVOLVE's tool usage feeds back into FORGE CORTEX's routing models, improving future model selection decisions.

5.5 ForgeEvent Types

FORGE EVOLVE emits three specialized event types into the unified `ForgeEvent` stream, enabling cross-subsystem observability:

These events enable real-time dashboards (via FORGE CONSOLE), automated alerting on modernization failures, and historical analysis of modernization patterns across projects. The `EQUIVALENCE_CHECK` events are particularly valuable for auditors, who can verify that behavioral equivalence was tested and confirmed for every migration unit.

6 Implementation

This section describes the technology stack, key implementation decisions, and deployment modes for FORGE EVOLVE.

6.1 Technology Stack

Table 6 summarizes the technology choices for each FORGE EVOLVE component.

Table 5: `ForgeEvent` types emitted by FORGE EVOLVE. Each event carries a standard header (timestamp, session ID, subsystem identifier) plus type-specific payload fields.

Event Type	Trigger	Payload
`MODERNIZATION_TASK`	Migration unit start, completion, or failure	Unit ID, tool assignment, source/target languages, LOC count, risk score, duration, outcome status
`EQUIVALENCE_CHECK`	Behavioral equivalence test execution	Module pair (legacy, modern), test count, pass rate, violations (if any), confidence bound, coverage metrics
`RULE_EXTRACTION`	Business rule identified	Rule ID, classification (calculation, validation, routing, constraint), source location, confidence score, RDF triple representation

Table 6: Technology stack for FORGE EVOLVE implementation. Each component uses mature, well-tested libraries to ensure reliability in production environments.

Component	Technology	Rationale
AST parsing	Tree-sitter	Incremental, 100+ languages, consistent AST structure
Dependency analysis	NetworkX	Mature graph algorithms (SCC, topological sort, shortest path)
Knowledge graphs	RDFLib + SPARQL	W3C-compliant RDF storage and query
Property-based testing	Hypothesis (Python)	Mature, shrinking, stateful testing
	fast-check (TS)	TypeScript-native property-based testing
AI tool integration	Anthropic SDK	Claude Code API access
	OpenAI SDK	Codex API access
	Google AI SDK	Gemini API access
	Cursor API	Local IDE integration
CLAR serialization	JSON-LD	Semantic web compatible, human-readable
Orchestration	Python + asyncio	Concurrent tool dispatch with backpressure
Execution sandboxing	Docker / gVisor	Isolated legacy and modern code execution
`ForgeEvent` emission	Protocol Buffers	Structured, versioned event serialization

6.2 Parser Architecture

The multi-language parsing infrastructure is built on Tree-sitter (Tree-sitter, 2023), which provides three critical capabilities for legacy code analysis:

Incremental parsing. Tree-sitter maintains a syntax tree that can be incrementally updated when source code changes, enabling interactive analysis during the transformation process. When the Transformation Engine modifies a module and the Validation Engine requests re-analysis, only the changed portions of the AST are re-parsed.

Error recovery. Legacy codebases frequently contain syntax that violates modern language standards (e.g., vendor-specific COBOL extensions, non-standard Fortran constructs). Tree-sitter's error-recovering parser continues to produce useful ASTs even when portions of the source code do not parse cleanly, annotating unparseable regions for manual review.

Query language. Tree-sitter's S-expression query language enables pattern matching across the AST, which FORGE EVOLVE uses extensively for business rule extraction. For example, the query `(if_statement condition: (comparison) consequence: (perform_statement))` matches COBOL conditional-PERFORM patterns that typically encode business routing rules.

For languages not natively supported by Tree-sitter (e.g., RPG, Natural/ADABAS), FORGE EVOLVE provides custom Tree-sitter grammars developed from the language specifications. Currently, custom

grammars exist for COBOL (ANSI-85 and IBM extensions), Fortran (77/90/95), Ada (83/95), and Visual Basic 6.

6.3 CLAR Implementation Details

The Cross-Language Abstract Representation is implemented as a typed property graph serialized in JSON-LD. Each CLAR node has a type drawn from a fixed vocabulary of 47 semantic node types organized into the four layers described in Section 4.2.4:

- **Control flow** (12 types): `SEQUENCE`, `BRANCH`, `FOR_LOOP`, `WHILE_LOOP`, `DO_UNTIL`, `SWITCH`, `TRY_CATCH`, `PARALLEL`, `PIPELINE`, `EVENT_HANDLER`, `STATE_MACHINE`, `COROUTINE`
- **Data flow** (15 types): `VARIABLE`, `CONSTANT`, `PARAMETER`, `RETURN_VALUE`, `ASSIGNMENT`, `ARITHMETIC`, `COMPARISON`, `CAST`, `AGGREGATE`, `COLLECTION`, `RECORD`, `FIXED_DECIMAL`, `FLOATING_POINT`, `STRING_OP`, `DATE_TIME`
- **Business logic** (10 types): `RULE`, `CONSTRAINT`, `INVARIANT`, `VALIDATION`, `CALCULATION`, `CLASSIFICATION`, `ROUTING`, `AUTHORIZATION`, `AUDIT`, `NOTIFICATION`
- **Infrastructure** (10 types): `FILE_IO`, `DB_QUERY`, `API_CALL`, `MESSAGE_SEND`, `MESSAGE_RECEIVE`, `STREAM_PROCESS`, `BATCH_JOB`, `TIMER`, `LOGGING`, `CONFIGURATION`

Source-to-CLAR projection is implemented as a Tree-sitter query-guided transformation: for each supported source language, a set of Tree-sitter queries identifies language-specific patterns and maps them to CLAR nodes. CLAR-to-target projection uses template-based code generation with language-specific code generators for each supported target language.

6.4 Deployment Modes

FORGE EVOLVE supports three deployment modes to accommodate diverse enterprise environments:

Cloud deployment. The standard deployment mode uses cloud-hosted AI tool APIs (Anthropic, OpenAI, Google) for transformation, with the orchestration layer running on customer-managed cloud infrastructure (AWS, Azure, GCP). Source code is transmitted to AI tool APIs for processing, making this mode suitable for non-classified, non-regulated codebases.

On-premises deployment. For organizations with data residency requirements or moderate security concerns, FORGE EVOLVE can be deployed entirely on-premises using locally hosted language models (e.g., Llama, Mistral, CodeLlama) as alternatives to cloud-hosted tools. The orchestration layer, parsing infrastructure, and validation engine run on customer hardware. Model quality may be lower than frontier cloud models, partially offset by domain-specific fine-tuning.

Air-gapped deployment. For classified environments (defense IL5/IL6, intelligence community), FORGE EVOLVE operates in a fully air-gapped mode with no external network connectivity. All language models are deployed locally on approved hardware, parsing and analysis tools run on isolated workstations, and `ForgeEvent` telemetry is stored locally rather than streamed. This mode requires pre-deployment model provisioning and periodic manual updates but provides full ITAR/CMMC compliance.

7 Experimental Evaluation

We evaluate FORGE EVOLVE on five dimensions: behavioral equivalence preservation, business rule extraction accuracy, tool orchestration effectiveness, migration velocity, and component contribution through ablation studies.

7.1 Benchmark Suite

Table 7 describes the benchmark suite used for evaluation. The suite comprises five language families, each with codebases of varying size and complexity drawn from open-source repositories, synthetic benchmarks, and anonymized industry partners.

The COBOL benchmarks include financial transaction processing, insurance claims, and payroll systems with extensive fixed-point arithmetic. The Fortran benchmarks cover scientific computing,

Table 7: Benchmark suite description. Each language family includes multiple codebases ranging from small modules to large systems. Business rules were manually annotated by domain experts to serve as ground truth for extraction evaluation.

Language	Codebases	Total LOC	Modules	Annotated Rules	Target
COBOL	8	1,247,000	892	1,847	Java 21
Fortran	6	486,000	341	623	Python 3.12
Ada	4	312,000	278	412	Rust 1.75
VB6	5	389,000	567	934	TypeScript 5.x
Legacy Java	7	823,000	1,204	1,156	Java 21 (modernized)
Total	**30**	**3,257,000**	**3,282**	**4,972**	—

Table 8: Behavioral equivalence pass rates (%) across five language families at three granularity levels. Results are computed over the full test suite (property-based + example-based) with tolerance ϵ set per the precision requirements of each domain. Standard deviations computed over 5 independent runs.

Language Family	Function-Level	Module-Level	System-Level
COBOL $\rightarrow$ Java	99.7 ± 0.1	99.4 ± 0.2	98.8 ± 0.3
Fortran $\rightarrow$ Python	99.3 ± 0.2	99.1 ± 0.3	98.2 ± 0.4
Ada $\rightarrow$ Rust	99.5 ± 0.1	99.2 ± 0.2	98.5 ± 0.3
VB6 $\rightarrow$ TypeScript	99.8 ± 0.1	99.6 ± 0.1	99.1 ± 0.2
Legacy Java $\rightarrow$ Modern Java	99.9 ± 0.0	99.7 ± 0.1	99.3 ± 0.2
Average	$\mathbf{99.6 \pm 0.1}$	$\mathbf{99.4 \pm 0.2}$	$\mathbf{98.8 \pm 0.3}$

engineering simulation, and signal processing. The Ada benchmarks represent real-time control systems and avionics software. The VB6 benchmarks include desktop business applications with GUI logic. The legacy Java benchmarks comprise J2EE enterprise applications, servlet-based web services, and EJB-based systems requiring modernization to Spring Boot and microservices.

All benchmarks include manually annotated business rules (4,972 total), created by domain experts who reviewed the source code and identified every business rule present. These annotations serve as the ground truth for evaluating the business rule extraction pipeline.

7.2 Behavioral Equivalence Results

Table 8 presents behavioral equivalence pass rates across the five language families at three levels of granularity: function/paragraph level, module level, and system level.

Several observations merit discussion:

Granularity degradation. Pass rates decrease from function to module to system level, consistent with Theorem 1's prediction that errors accumulate through module composition. The system-level degradation is modest (average 0.8 percentage points below function-level), validating the theorem's bound under the assumption that most inter-module interactions are interface-preserving.

Language-specific patterns. COBOL $\rightarrow$ Java and Fortran $\rightarrow$ Python exhibit the lowest system-level pass rates due to precision-sensitive computations (COBOL fixed-point arithmetic, Fortran double-precision floating-point) that require careful type mapping. Legacy Java $\rightarrow$ Modern Java achieves the highest rates because the source and target share the same type system, eliminating precision concerns.

Failure analysis. The 1.2% system-level failure rate in COBOL $\rightarrow$ Java is attributable to three root causes: (i) EBCDIC-to-Unicode character encoding edge cases in string comparisons (0.5%), (ii) fixed-point rounding differences in multi-step financial calculations (0.4%), and (iii) COBOL file status code handling that has no direct Java equivalent (0.3%). All failures were identified by the Validation Engine and flagged for manual review.

Figure 8: Behavioral equivalence pass rate as a function of test suite size (number of property-based test inputs). Pass rates converge rapidly: 95% of the final pass rate is achieved within 10,000 tests, and 99% within 50,000 tests. The dashed line shows the confidence bound from Theorem 3.

Table 9: Business rule extraction performance (precision, recall, F1) by rule type and language family. Results are macro-averaged across codebases within each language family.

Language	Rule Type	Precision	Recall	F1
COBOL	Calculation	0.93	0.89	0.91
	Validation	0.91	0.87	0.89
	Routing	0.88	0.82	0.85
	Constraint	0.86	0.79	0.82
Fortran	Calculation	0.91	0.88	0.89
	Validation	0.87	0.83	0.85
	Routing	0.83	0.78	0.80
	Constraint	0.85	0.81	0.83
Ada	Calculation	0.92	0.90	0.91
	Validation	0.90	0.86	0.88
	Routing	0.87	0.83	0.85
	Constraint	0.89	0.85	0.87
VB6	Calculation	0.94	0.91	0.92
	Validation	0.92	0.88	0.90
	Routing	0.89	0.85	0.87
	Constraint	0.87	0.82	0.84
Legacy Java	Calculation	0.95	0.93	0.94
	Validation	0.93	0.90	0.91
	Routing	0.91	0.87	0.89
	Constraint	0.90	0.86	0.88
Overall Average		**0.90**	**0.86**	**0.87**

7.3 Business Rule Extraction Results

Table 9 presents business rule extraction performance measured by precision, recall, and F1 score against the manually annotated ground truth.

Calculation rules achieve the highest F1 scores (0.89–0.94) because they follow recognizable arithmetic patterns that AST analysis can reliably identify. Routing rules achieve the lowest scores (0.80–0.89) because they often involve complex conditional logic spread across multiple code sections, requiring the LLM interpretation phase to synthesize context from disparate code fragments.

7.4 Tool Orchestration Effectiveness

Table 10 compares the orchestrated multi-tool approach against single-tool baselines. For each baseline, all tasks are assigned to the specified tool regardless of task features.

The orchestrated approach achieves the highest quality (99.4%) at a cost comparable to the cheapest single-tool baseline (Codex at $2.80/kLOC vs. orchestrated at $2.90/kLOC). This is because the orchestrator routes low-complexity tasks to cheaper tools (Codex for batch translation) while reserving expensive high-capability tools (Claude Code) for complex tasks where their advantage justifies the cost. The 23% quality improvement over the best single-tool baseline (Claude Code at 97.8%) demonstrates that tool-task matching provides substantial benefits even compared to the strongest individual tool.

Table 10: Tool selection effectiveness. The orchestrated approach selects the optimal tool per task, compared against single-tool baselines. Quality is measured as the average behavioral equivalence pass rate weighted by module complexity. Cost is total API cost in USD per 1,000 LOC.

Approach	Quality (%)	Cost ($/kLOC)	Time (min/kLOC)
Claude Code only	97.8	4.20	8.3
Codex only	95.1	2.80	5.7
Gemini only	94.6	3.50	6.2
Cursor only	93.2	3.10	12.4
Orchestrated (ours)	**99.4**	**2.90**	**6.8**

Table 11: Migration velocity comparison. Manual baselines are from published industry studies and represent typical velocities for experienced migration teams. FORGE EVOLVE velocities include all phases (discovery, planning, transformation, validation).

Metric	Manual Baseline	FORGE EVOLVE	Speedup
Lines migrated per day	200–500	1,200–3,500	4.5×–7.0×
Modules completed per week	2–5	12–35	5.0×–7.0×
Discovery phase (per 100K LOC)	2–4 weeks	1–3 days	6.7×–9.3×
Planning phase (per 100K LOC)	3–6 weeks	2–5 days	8.4×–10.5×
Validation phase (per module)	1–3 days	2–8 hours	3.0×–9.0×
End-to-end (100K LOC)	6–12 months	4–8 weeks	6.5×–6.0×
Average speedup	—	—	**6.3×**

7.5 Migration Velocity

Table 11 presents migration velocity metrics comparing FORGE EVOLVE against manual migration baselines from published industry studies.

The discovery and planning phases show the largest speedups (6.7–10.5×) because they are most amenable to automation: AST analysis, dependency mapping, and risk scoring are computationally intensive but algorithmically well-defined. The validation phase shows the smallest speedup (3.0–9.0×) because behavioral equivalence checking requires executing both legacy and modern code, which is constrained by runtime execution speed rather than analysis speed.

The end-to-end speedup of approximately 6.3× translates to 60–70% time reduction: a modernization program that would require 12 months manually can be completed in approximately 8 weeks with FORGE EVOLVE.

7.6 Ablation Studies

Table 12 presents ablation results measuring the contribution of each FORGE EVOLVE component to overall performance. Each ablation removes one component and replaces it with a simple baseline.

The ablation results reveal several findings:

Tool orchestration is the single most impactful component for behavioral equivalence, with its removal reducing pass rates by 3.7 percentage points. This confirms that intelligent tool selection is not merely a cost optimization but a quality-critical capability.

CLAR contributes 1.6 percentage points to behavioral equivalence by enabling more accurate cross-language type mapping and preserving semantic intent through the intermediate representation. Its removal also reduces migration velocity by 11% because direct translation requires more iterations to resolve type and precision issues.

Property-based testing contributes 1.3 percentage points to behavioral equivalence while slightly *reducing* velocity (because comprehensive testing takes time). This trade-off is favorable: the quality gain outweighs the modest velocity cost.

Figure 9: Distribution of migration units by risk score and complexity (measured in cyclomatic complexity) for the full benchmark suite. The majority of units (68%) fall in the low-risk, low-complexity quadrant and can be migrated automatically with minimal human oversight. High-risk, high-complexity units (7%) require dedicated human review and are migrated last.

Figure 10: Impact of migration ordering on cumulative system-level equivalence pass rate. Risk-ordered migration (low-risk first) achieves stable high pass rates early and maintains them as higher-risk units are added. Random ordering exhibits significant variance and lower early-stage pass rates. Reverse ordering (high-risk first) incurs early failures that propagate through dependent modules.

LLM interpretation is critical for business rule extraction, contributing 15 F1 points. AST-only extraction identifies structural patterns but cannot interpret their business semantics, leading to many false positives (patterns that look like rules but are not) and false negatives (rules expressed in non-standard patterns).

Validation Engine removal produces the highest velocity (3,100 LOC/day) but the lowest quality (96.3% BE pass rate), illustrating the fundamental quality-velocity trade-off that FORGE EVOLVE navigates.

8 Case Studies

We present four case studies demonstrating FORGE EVOLVE in representative industry verticals. Case study details are anonymized per client confidentiality agreements; quantitative results are reported with client approval.

8.1 Financial Services: COBOL to Cloud

Context. A mid-tier U.S. bank operates a core banking platform comprising 2.1 million lines of COBOL running on an IBM z/OS mainframe. The system processes 4.2 million transactions daily, implementing over 3,200 business rules governing interest calculations, fee structures, regulatory reporting (SOX, FINRA), and fraud detection. The bank's three remaining COBOL developers are within five years of retirement, and the mainframe licensing costs exceed $8M annually.

Discovery. FORGE EVOLVE's Discovery Engine analyzed the full 2.1M LOC codebase in 4.3 days. Static analysis identified 1,847 modules with an average cyclomatic complexity of 18.7 (high, reflecting decades of incremental modifications). The dependency graph revealed 12 strongly connected components, the largest containing 147 mutually dependent modules centered on the general ledger subsystem. Business rule extraction identified 3,247 rules (2,891 calculation rules, 234 validation rules, 89 routing rules, 33 constraint rules) with an estimated F1 of 0.88 against a 500-rule validation sample reviewed by the bank's senior COBOL developer. The cryptographic inventory identified 47 uses of DES and 3DES encryption—quantum-vulnerable algorithms flagged for FORGE QBIT migration.

Planning. The Migration Planner decomposed the codebase into 156 migration units with a risk-scored ordering that placed the customer statement generation subsystem (low risk, high business value, minimal dependencies) first and the general ledger SCC (highest risk) last. The total plan spanned 28 weeks, with the first production deployment targeted at week 6.

Transformation. Tool orchestration assigned 62% of tasks to Codex (batch COBOL-to-Java translation for straightforward modules), 24% to Claude Code (complex modules requiring multi-file context and business rule preservation), 8% to Gemini (report generation modules with large copybook hierarchies), and 6% to Cursor (interactive refinement of UI-adjacent modules). The target stack was Java 21 with Spring Boot, PostgreSQL, and Apache Kafka for event streaming.

Results. Module-level behavioral equivalence: 99.3% pass rate. System-level equivalence after integration testing: 98.7%. The 1.3% system-level failures were attributable to EBCDIC sort-order differences in customer name comparisons (resolved by implementing EBCDIC-compatible collation in Java). Migration velocity: 1,850 LOC/day average (6.2$\times$ faster than the bank's previous manual migration attempt, which had been abandoned at 300 LOC/day). Total project duration: 24 weeks

Table 12: Ablation study results. Each row removes one component and measures the impact on behavioral equivalence (BE) pass rate, business rule extraction F1, and migration velocity (LOC/day). Baselines: random tool selection, no CLAR (direct translation), no property-based testing, no risk scoring (alphabetical ordering), no LLM interpretation (AST-only rule extraction).

Configuration	BE Pass (%)	Rule F1	LOC/day
Full FORGE EVOLVE	99.4	0.87	2,350
− Tool orchestration (random)	95.7	0.87	1,890
− CLAR (direct translation)	97.8	0.85	2,100
− Property-based testing	98.1	0.87	2,450
− Risk-scored ordering	98.6	0.87	2,180
− LLM interpretation	99.2	0.72	2,340
− Validation Engine	96.3	0.87	3,100

Figure 11: Cost scaling analysis for FORGE EVOLVE as a function of codebase size (LOC). Total cost includes AI tool API charges, compute infrastructure, and validation overhead. The cost-per-LOC decreases with codebase size due to amortization of discovery and planning costs, reaching a plateau around 500K LOC. Dotted lines show manual migration cost estimates for comparison.

(vs. the estimated 3–5 years for manual migration). Mainframe licensing savings: $8.2M/year. Extracted business rules fed into FORGE MEMORY, creating the bank's first comprehensive digital documentation of its core banking logic.

8.2 Defense: Ada/Fortran Modernization

Context. A defense contractor maintains a tactical mission planning system comprising 480,000 lines of Ada 83 and 120,000 lines of Fortran 77, running on specialized real-time hardware. The system computes terrain analysis, route optimization, and threat assessment for ground vehicle operations. ITAR restrictions require air-gapped development; no source code may traverse the network boundary.

Approach. FORGE EVOLVE was deployed in air-gapped mode with locally hosted CodeLlama-34B and Mistral-7B as the primary AI tools (replacing cloud-hosted Claude Code and Codex). A custom Tree-sitter grammar was developed for the Ada 83 dialect used in the system, which predates the standardized Ada 95 features.

Discovery. Analysis completed in 2.8 days, identifying 278 Ada modules and 63 Fortran subroutines. Dependency analysis revealed a clean layered architecture (a positive legacy of the original Ada-mandated design discipline) with no circular dependencies. Business rule extraction identified 412 rules, predominantly calculation rules implementing tactical planning algorithms.

Transformation. The target was Rust 1.75, selected for its combination of memory safety, real-time capabilities, and modern tooling. Ada's range types and constraint checking mapped naturally to Rust's type system and trait-based generics. Fortran numerical routines were translated to Rust with SIMD optimization where applicable.

Results. Module-level equivalence: 99.5%. Real-time performance: the modernized Rust system met all timing constraints with 15% margin (compared to 3% margin in the original Ada/Fortran system), attributable to Rust's zero-cost abstractions and modern compiler optimizations. The air-gapped deployment mode functioned correctly, with locally hosted models achieving 94% of the quality of cloud-hosted frontier models. Total duration: 14 weeks for the full 600K LOC codebase.

8.3 Healthcare: HL7 to FHIR

Context. A regional hospital network operates a clinical data integration platform built on HL7 v2 messaging, implemented in a combination of legacy Java (J2EE era) and custom C interface adapters totaling 340,000 LOC. The platform must migrate to FHIR R4 to comply with the 21st Century Cures Act interoperability requirements and to integrate with modern EHR systems.

Table 13: Market segment analysis summarizing the four case studies. Entry product indicates the recommended starting engagement.

Segment	Codebase Size	Duration	BE Pass Rate	Key Challenge
Financial Services	2.1M LOC	24 weeks	98.7%	Fixed-point precision; EBCDIC encoding
Defense	600K LOC	14 weeks	99.5%	Air-gapped deployment; real-time constraints
Healthcare	340K LOC	11 weeks	99.1%	Data model transformation; HIPAA compliance
Enterprise IT	1.4M LOC	18 weeks	99.3%	Monolith decomposition; session state

Challenge. HL7 v2 to FHIR migration is not a simple code translation—it requires transforming the data model from segment-based flat messages to resource-based hierarchical documents. Business rules embedded in the HL7 processing logic must be preserved while fundamentally changing the data representation.

Approach. FORGE EVOLVE's CLAR representation was particularly valuable here: the business logic layer captured the clinical rules (e.g., "if patient age > 65 and medication class = opioid, require pharmacist review") independently of the HL7/FHIR data representation, allowing the rules to be faithfully re-implemented in the FHIR context.

Results. Business rule preservation: 97.8% of clinical rules were correctly migrated (verified against manual expert review). The 2.2% unpreserved rules were FHIR-incompatible HL7 workarounds that clinical staff confirmed were no longer needed. HIPAA compliance: all PHI handling was preserved in the modernized code, verified through dedicated HIPAA-focused test suites. Duration: 11 weeks.

8.4 Enterprise IT: Monolith Decomposition

Context. A Fortune 500 manufacturing company operates an enterprise resource planning (ERP) system built as a Java 6 monolith with 1.4 million LOC, deployed on JBoss Application Server. The system uses EJB 2.x entity beans, JSP-based UI, and a single Oracle database with 847 tables. The modernization goal is decomposition into Spring Boot microservices with React frontend and per-service PostgreSQL databases.

Discovery. Analysis identified 1,204 modules (Java classes) with an average of 23.7 dependencies per class—a high coupling metric indicating significant architectural decay. The dependency graph contained 8 large SCCs, the largest spanning 312 classes in the order management subsystem. Business rule extraction identified 1,156 rules, many of which were scattered across multiple classes in violation of separation of concerns.

Planning. The Migration Planner identified 18 natural service boundaries through community detection (Blondel et al., 2008) on the dependency graph, aligning with the company's business domain model. Database tables were assigned to services based on access patterns extracted from data flow analysis. The migration was sequenced to deliver the product catalog service (read-heavy, minimal state, low risk) first, building team confidence before tackling the order management monolith.

Transformation. Tool assignments: Claude Code (38%, architecture refactoring and complex business logic), Codex (31%, EJB-to-Spring conversion), Cursor (22%, interactive JSP-to-React UI migration), Gemini (9%, database schema extraction and documentation). The transformation generated 18 Spring Boot microservices, 18 PostgreSQL schemas, API gateway configuration (Kong), and service mesh configuration (Istio).

Results. Module-level equivalence: 99.3%. System integration testing: 98.9% pass rate, with failures concentrated in session state handling during the EJB-to-REST transition (resolved by implementing distributed session management with Redis). Performance: the microservices architecture achieved $3.2\times$ throughput improvement on the order processing pipeline due to independent scaling of

Table 14: Competitive comparison of FORGE EVOLVE against alternative modernization approaches. Quality refers to behavioral equivalence pass rates. Tool flexibility indicates whether multiple AI tools are supported. Knowledge preservation indicates structured extraction and documentation of business rules.

Approach	Quality	Velocity	Tool Flex.	Knowledge	Verification
Manual rewrite	Medium	Low	N/A	Low	Manual
Single-tool AI	High	High	No	Low	Partial
TransCoder-style	Medium	High	No	None	None
FORGE EVOLVE	Very High	Very High	Yes	High	Automated

Figure 12: End-to-end FORGE EVOLVE methodology flow. The five phases—Discover, Plan, Migrate, Validate, Operate—correspond to the five architecture modules. Feedback loops enable iterative refinement: validation failures feed back to transformation, and operational insights from the Operate phase inform future discovery. The timeline shows typical durations for a 500K LOC codebase.

bottleneck services. Duration: 18 weeks (vs. the company's initial estimate of 2–3 years for manual decomposition).

9 Discussion

9.1 Key Findings

Our evaluation and case studies yield five principal findings:

Finding 1: Tool orchestration is quality-critical, not just cost-optimal. The ablation study (Table 12) demonstrates that intelligent tool selection contributes 3.7 percentage points to behavioral equivalence—the largest single-component contribution. This challenges the common assumption that AI tool selection is primarily a cost optimization problem; in the modernization context, selecting the right tool for each task is the most important quality lever.

Finding 2: Behavioral equivalence composes well in practice. Theorem 1 predicts that module-level equivalence composes to system-level equivalence with bounded error accumulation. Our experimental results confirm this prediction: the average system-level degradation is only 0.8 percentage points below function-level, indicating that inter-module error propagation is modest when interfaces are preserved. This validates the incremental migration strategy and provides assurance that organizations can adopt FORGE EVOLVE for module-by-module migration without risking catastrophic system-level failures.

Finding 3: LLM interpretation is essential for business rule extraction. AST-based structural analysis alone achieves only 0.72 F1 on business rule extraction (Table 12), compared to 0.87 with LLM interpretation. The 15 F1-point improvement comes from the LLM's ability to interpret business *intent* from code patterns that are structurally ambiguous—a capability that pure program analysis cannot provide. This finding has implications beyond modernization: any system that seeks to extract domain knowledge from code will benefit from combining structural analysis with language model interpretation.

Finding 4: Air-gapped deployment is viable with acceptable quality trade-offs. The defense case study demonstrates that FORGE EVOLVE can operate effectively in air-gapped environments using locally hosted models, achieving 94% of cloud-hosted quality. The 6% quality gap is concentrated in complex architectural refactoring tasks where frontier models' superior reasoning capabilities provide the greatest advantage. For classified environments where cloud deployment is impossible, this trade-off is acceptable.

Finding 5: The DNA metaphor is operationally validated. Across all four case studies, FORGE EVOLVE's outputs directly enabled other FORGE OS subsystem adoption. The financial services client used extracted business rules in FORGE MEMORY within two weeks of modernization completion. The defense client's crypto inventory was immediately actionable by FORGE QBIT.

This validates the positioning of FORGE EVOLVE as the prerequisite—the "DNA"—for enterprise AI adoption through FORGE OS.

9.2 Limitations

FORGE EVOLVE has several limitations that warrant discussion:

Behavioral equivalence is statistical, not formal. Despite Theorem 3's confidence bounds, our behavioral equivalence guarantees are based on testing rather than formal proofs. For safety-critical systems (e.g., avionics, nuclear control), formal verification of critical code paths may be necessary as a complement to FORGE EVOLVE's testing-based approach. We view this as a graduated assurance model: FORGE EVOLVE provides broad automated testing, and formal verification can be applied surgically to the highest-criticality components.

Language coverage. While FORGE EVOLVE supports five language families, significant legacy languages remain unsupported, including RPG (IBM midrange systems), Natural/ADABAS (mainframe 4GL), and PL/I. Each new source language requires a Tree-sitter grammar and source-to-CLAR projection rules—approximately 2–4 weeks of development effort per language.

Tool capability evolution. AI coding tools are improving rapidly. The capability profiles and tool selection criteria in Tables 2 and 10 reflect the state of the art as of early 2026. Tool capabilities may shift significantly within months, requiring periodic re-calibration of the orchestrator's models. The Thompson Sampling approach mitigates this through continuous online learning, but major tool releases may cause temporary suboptimality.

Human-in-the-loop requirements. Despite high automation rates, FORGE EVOLVE requires human expertise at several points: validation of extracted business rules against domain knowledge, review of high-risk migration units, and resolution of behavioral equivalence failures that automated analysis cannot diagnose. The system augments rather than replaces human expertise.

Scalability limits. Our largest benchmarked codebase is 2.1M LOC. Codebases exceeding 10M LOC may encounter scalability challenges in dependency graph analysis (graph algorithms scale as $O(V + E)$ but with large constants for dense graphs) and in maintaining coherent CLAR representations across thousands of modules. Hierarchical decomposition strategies could address this, but have not been validated.

9.3 Ethical Considerations

Workforce impact. Legacy modernization automation may displace manual migration teams—typically junior-to-mid-level developers assigned to tedious rewrite work. However, our case studies suggest that FORGE EVOLVE shifts the human role from manual translation to higher-value activities: business rule validation, architectural decision-making, and quality assurance. Organizations deploying FORGE EVOLVE should invest in retraining programs that prepare migration teams for these supervisory roles.

Intellectual property. When cloud-hosted AI tools process proprietary source code, IP concerns arise. FORGE EVOLVE's air-gapped deployment mode addresses this for the most sensitive environments. For cloud deployments, organizations must evaluate each tool vendor's data retention and training policies. FORGE EVOLVE's tool-agnostic design allows organizations to exclude specific vendors based on IP policy.

Accountability for failures. When an AI-assisted modernization introduces a behavioral regression that reaches production, accountability is unclear. FORGE EVOLVE's governance artifact trail (Table 4) provides the evidentiary chain needed to attribute failures to specific decisions, tools, and validation gaps, supporting post-incident analysis and accountability.

Bias in tool selection. The Thompson Sampling approach for tool selection could develop biased capability estimates if the initial evaluation set is unrepresentative. To mitigate this, FORGE EVOLVE enforces a minimum exploration rate (5% of tasks assigned to non-preferred tools) and periodically resets capability estimates when major tool updates are released.

10 Conclusion

Legacy code modernization is the prerequisite for enterprise AI adoption. Organizations cannot build knowledge graphs from undocumented COBOL, cannot deploy quantum-safe cryptography in Fortran subroutines, cannot run autonomous systems on Ada 83 control logic, and cannot fine-tune domain models on spaghetti code. Until the code is modernized, the organization is locked out of the AI era.

FORGE EVOLVE addresses this modernization imperative through five integrated capabilities: multi-agent tool orchestration that selects the optimal AI coding tool per task, behavioral equivalence verification that guarantees functional preservation, automated business rule extraction that surfaces implicit domain knowledge, complexity-aware migration planning that sequences transformations for minimum risk, and the Cross-Language Abstract Representation that decouples source analysis from target generation.

Our evaluation demonstrates that FORGE EVOLVE achieves behavioral equivalence pass rates exceeding 99% at the module level and 98.8% at the system level, business rule extraction F1 scores of 0.87, and migration velocity improvements of 6.3× over manual baselines—across five legacy language families and four industry verticals. The theoretical foundations—equivalence composability, decomposition hardness, and confidence bounds—provide formal assurance that the empirical results rest on sound mathematical principles.

As the "DNA" of FORGE OS, FORGE EVOLVE transforms an organization's most neglected software assets into the foundation for modern AI capabilities. Extracted business rules flow into FORGE MEMORY's knowledge graph, cryptographic inventories feed FORGE QBIT's post-quantum migration queue, modernized control logic targets FORGE KINETIC's autonomous deployment, and clean codebases enable FORGE CORTEX's domain-specific fine-tuning. Modernization is not the destination—it is the first step.

10.1 Future Work

Several directions merit future investigation:

Expanded language support. Priority additions include RPG (for IBM iSeries modernization), PL/I (for mainframe scientific applications), and Perl (for legacy web infrastructure). Each requires a Tree-sitter grammar and CLAR projection rules.

Federated modernization. For organizations with codebases distributed across multiple sites (common in defense and multinational enterprises), a federated FORGE EVOLVE deployment would enable parallel modernization with cross-site consistency checking while respecting data sovereignty constraints.

Continuous evolution. Beyond one-time modernization, FORGE EVOLVE could serve as a continuous code evolution engine that proactively identifies and addresses technical debt as it accumulates, preventing the next generation of legacy code from forming.

Formal verification integration. Integrating lightweight formal verification (e.g., bounded model checking, abstract interpretation) for critical code paths would strengthen the behavioral equivalence guarantee from statistical to formal for the highest-criticality modules.

LLM-generated CLAR projections. Currently, source-to-CLAR projections are implemented as hand-coded Tree-sitter query transformations. Fine-tuning a code model to generate CLAR projections from arbitrary source languages could dramatically reduce the effort required to add new language support.

References

R. Khadka et al. How do professionals perceive legacy systems and software modernization? In *Proc. ICSE*, pages 36–47, June 2014.

R. Seacord, D. Plakosh, and G. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, 2003.

Gartner Research. Legacy application modernization: A strategic approach. Technical report, Gartner Inc., 2024.

Stripe & Harris Poll. The Developer Coefficient: How software engineering talent transforms business. Technical report, Stripe, 2018.

Standish Group. The CHAOS Report 2020: Beyond Infinity. Technical report, Standish Group International, 2020.

Micro Focus. COBOL: The state of the language in 2020. Whitepaper, Micro Focus International, 2020.

M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan Kaufmann, 1995.

J. Bisbal et al. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, 1999.

B. Wu et al. The butterfly methodology: A gateway-free approach for migrating legacy information systems. In *Proc. ICECCS*, pages 200–205, 1997.

S. Comella-Dorda et al. A survey of legacy system modernization approaches. Technical Report CMU/SEI-2000-TN-003, Software Engineering Institute, 2000.

M. Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, July 2021.

Y. Li et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022.

Anthropic. Claude: A family of large language models. Technical report, Anthropic, 2024.

Google DeepMind. Gemini: A family of highly capable multimodal models. Technical report, Google, 2024.

Anysphere. Cursor: The AI-first code editor. `https://cursor.com`, 2024.

B. Rozière, M.-A. Lachaux, L. Chanussot, and G. Lample. Unsupervised translation of programming languages. In *Proc. NeurIPS*, volume 33, December 2020.

R. Pan et al. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proc. ICSE*, pages 866–878, April 2024.

C. E. Jimenez et al. SWE-bench: Can language models resolve real-world GitHub issues? In *Proc. ICLR*, May 2024.

Cognition AI. Introducing Devin, the first AI software engineer. Blog post, March 2024.

P. Gauthier. Aider: AI pair programming in your terminal. GitHub repository, 2024.

Tree-sitter Contributors. Tree-sitter: An incremental parsing system for programming tools. `https://tree-sitter.github.io`, 2023.

I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proc. MSR*, pages 1–5, May 2005.

K. Ali and O. Lhoták. Application-only call graph construction. In *Proc. ECOOP*, pages 688–712, June 2012.

L. Moonen. Generating robust parsers using island grammars. In *Proc. WCRE*, pages 13–22, October 2001.

T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL*, pages 49–61, January 1995.

B. Cornelissen et al. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Engineering*, 35(5):684–702, 2009.

C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

H. Boomsma, B. Gross, and A. van Deursen. Dead code elimination in web applications. In *Proc. SCAM*, pages 164–168, September 2012.

A. Hagberg, P. Swart, and D. Chult. Exploring network structure, dynamics, and function using NetworkX. In *Proc. SciPy*, pages 11–15, 2008.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP*, pages 268–279, September 2000.

D. R. MacIver, Z. Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Engineering*, 37(5):649–678, 2011.

L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

D. Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Software Engineering and Methodology*, 11(2):256–290, 2002.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, pages 348–370, April 2010.

C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, December 2008.

W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. A transformer-based approach for source code summarization. In *Proc. ACL*, pages 4998–5007, July 2020.

A. LeClair, S. Jiang, and C. McMillan. A neural model for generating natural language summaries of program subroutines. In *Proc. ICSE*, pages 795–806, May 2019.

H. M. Sneed. Extracting business rules from source code. In *Proc. WPC*, pages 240–247, July 2001.

V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet. A model-driven reverse engineering framework for extracting business rules out of a Java application. In *Proc. RuleMl*, pages 17–31, July 2013.

D. Kang, J. Xu, and Y. Xu. Ontology learning from source code based on class hierarchies. In *Proc. JIST*, pages 130–145, December 2012.

D. Ratiu, R. Marinescu, and J. Jurjens. The logical modularity of programs. In *Proc. WCRE*, pages 123–132, November 2004.

B. P. Gerkey and M. J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9):939–954, 2004.

T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohmé. Coalition structure generation with worst case guarantees. *Artificial Intelligence*, 111(1–2):209–238, 1999.

Q. Wu et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, August 2023.

S. Hong et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proc. ICLR*, May 2024.

O. Goldschmidt and D. S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Mathematics of Operations Research*, 19(1):24–37, 1994.

T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3–4):285–294, 1933.

V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*, 2008(10):P10008, 2008.

577 Industries R&D Lab. FORGE Cortex: An Agent-Legible Intelligence Architecture for Model-Agnostic Enterprise AI. Technical report, 577 Industries Incorporated, 2025.

577 Industries R&D Lab. FORGE QBit: Heterogeneous Post-Quantum Cryptography and Physics-Informed Computation for Sovereign AI Systems. Technical report, 577 Industries Incorporated, 2025.

577 Industries R&D Lab. FORGE Memory: Governed Multi-Agent Orchestration with Predictive Human-in-the-Loop Intelligence. Technical report, 577 Industries Incorporated, 2025.

577 Industries R&D Lab. FORGE Kinetic: Autonomous Multi-Agent Swarm Intelligence for Sovereign Edge Deployment. Technical report, 577 Industries Incorporated, 2025.