Synthesis: A Federated Capability Ecosystem for Safe AI Self-Extension Through Test-Driven Development and Graduated Trust

Anthony Maio
Independent Researcher
https://orcid.org/0009-0003-4541-8515
anthony@making-minds.ai
https://making-minds.ai

February 2026

Abstract

As AI agents become more capable, there is increasing interest in systems that can extend their own capabilities through code generation and tool use. However, naive code generation approaches produce unreliable outputs that may fail silently, introduce security vulnerabilities, or behave unexpectedly—a phenomenon well-documented in evaluations of large language model code generation [Chen et al., 2021, Liu et al., 2023]. We present SYNTHESIS, a federated capability ecosystem for safe AI self-extension that addresses these challenges through three integrated mechanisms: (1) **Test-Driven Synthesis**, where comprehensive test suites are generated before implementation code and capabilities must pass all tests before deployment; (2) **Graduated Trust**, where newly synthesized capabilities start in maximally restricted sandboxes and progressively earn privileges through demonstrated reliability across quantified thresholds; and (3) **Composition Over Creation**, where the system exhaustively searches a shared LIVE EXCHANGE and attempts to compose existing verified capabilities before synthesizing new code, creating network effects that benefit all participating agents. The architecture includes a trust bootstrapping protocol that solves the cold-start problem for new deployments through founding validators and pre-verified seed capabilities. Our empirical measurements show realistic success rates (50–70% one-shot, 70–85% after iterative refinement) while maintaining honest metrics about system limitations. SYNTHESIS provides a foundation for AI systems that can safely adapt to new requirements without compromising reliability, security, or auditability.

1 Introduction

The ability to extend one's own capabilities in response to novel requirements represents a significant step toward more autonomous AI systems. Recent advances in large language model (LLM) code generation have demonstrated impressive capabilities on programming benchmarks [Chen et al., 2021, Austin et al., 2021, Hendrycks et al., 2021], yet these same evaluations reveal a critical gap: generated code that appears syntactically correct often contains subtle logical flaws, unhandled edge cases, or security vulnerabilities [Liu et al., 2023].

Consider an AI agent that encounters a task requiring functionality it does not possess. The naive approach—generating code and immediately deploying it—introduces several documented risks:

- **Silent failures**: Generated code may produce incorrect outputs without raising errors, as demonstrated by studies showing that superficially correct code frequently fails on edge cases not present in limited test suites [Liu et al., 2023]

- **Security vulnerabilities**: Untested code may access unauthorized resources, expose sensitive data, or contain injection vulnerabilities that static analysis alone cannot reliably detect [Li et al., 2024, Charoenwet et al., 2024]
- **Unreliable behavior**: Edge cases, boundary conditions, and error handling are often inadequate in generated code, as even state-of-the-art models achieve only 70–85% pass rates on straightforward programming tasks [Chen et al., 2021]
- **Cascading errors**: In multi-agent systems, faulty capabilities may corrupt downstream processes, amplifying the impact of individual failures [Tran et al., 2025]

These challenges are compounded when AI systems operate autonomously. Unlike human-supervised code review, autonomous agents must validate their own generated code without external verification—a fundamental tension between capability extension and safety that has been identified as a core challenge in AI alignment [Amodei et al., 2016].

We propose SYNTHESIS, a federated capability ecosystem that addresses these challenges through three integrated principles:

Test-Driven Development. Before generating any implementation code, SYNTHESIS creates comprehensive test suites based on capability requirements. Implementation is then iteratively refined until all tests pass. This approach, grounded in decades of software engineering research showing TDD produces more reliable software with fewer defects [Beck, 2003, Williams et al., 2003, emp, 2016], ensures that generated capabilities are demonstrably correct rather than merely plausible-looking.

Graduated Trust. Every newly synthesized capability starts in a maximally restricted sandbox with no network access, no filesystem access, and strict resource limits. As capabilities demonstrate reliability through successful executions across quantified thresholds, they progressively earn expanded privileges. This mirrors how trust and reputation systems operate in distributed networks [Kamvar et al., 2003, Jøsang et al., 2007, Granatyr et al., 2015], adapting these insights to the context of executable code modules.

Composition Over Creation. Before synthesizing new code, SYNTHESIS exhaustively searches a shared LIVE EXCHANGE for existing verified capabilities and attempts to compose them into solutions using various orchestration strategies [Milanovic and Malek, 2004, Zhang et al., 2025]. Synthesis becomes the fallback rather than the default, reducing attack surface by preferring proven implementations and creating network effects where every agent that contributes to the ecosystem makes it more capable for all others.

The remainder of this paper is organized as follows. Section 2 reviews related work on code generation, AI safety, and trust systems. Section 3 details the SYNTHESIS architecture including the LIVE EXCHANGE, composition engine, and TDD synthesizer. Section 4 presents the graduated trust system, scoring mechanisms, and bootstrapping protocol. Section 5 provides honest metrics on synthesis success rates. Section 6 discusses limitations, philosophical foundations, and future directions. Section 7 summarizes our contributions.

2 Related Work

2.1 Code Generation with Large Language Models

Large language models have demonstrated impressive code generation capabilities across multiple benchmarks. Codex [Chen et al., 2021] achieved 28.8% pass@1 on HumanEval (164 hand-crafted programming problems), reaching 70.2% pass@100 with multiple samples. Subsequent work

introduced additional benchmarks: MBPP with 974 basic programming problems [Austin et al., 2021], APPS with 10,000 competitive programming problems [Hendrycks et al., 2021], and CodeXGLUE with 10 tasks across 14 datasets [Lu et al., 2021].

However, rigorous evaluation reveals significant limitations. Liu et al. [2023] demonstrated that existing benchmarks underestimate failure rates by 19–28.9% due to insufficient test cases—many solutions that pass HumanEval fail on edge cases and boundary conditions. This finding motivates our test-first approach: rather than trusting generated code that passes limited tests, we generate comprehensive tests before implementation and iterate until all pass.

Code understanding models like CodeBERT [Feng et al., 2020] provide foundations for semantic code search, enabling our capability matching and composition features. These models support finding functionally equivalent code across different implementations, a key requirement for our composition engine.

2.2 Test-Driven Development

Test-driven development (TDD) is a software methodology where tests are written before implementation code [Beck, 2003]. Empirical studies demonstrate that TDD produces more reliable software with fewer defects, though effect sizes vary by context [Williams et al., 2003, emp, 2016]. Property-based testing extends TDD by generating test cases from specifications rather than examples [Goldstein et al., 2024].

Recent work has applied TDD principles to LLM code generation. Meta AI [2024] demonstrated that LLM-generated test improvements achieved 73% acceptance rates at Meta, with 25% coverage improvements. Execution-guided synthesis [Chen et al., 2019] uses test execution feedback to guide program generation, while DeepCoder [Balog et al., 2017] combines neural networks with enumerative search guided by input-output examples.

Our approach extends this line of work by making TDD integral to a complete capability lifecycle, from generation through graduated deployment, rather than a standalone code generation technique.

2.3 Sandboxing and Container Security

The challenge of safely executing untrusted code has been addressed through various isolation techniques. Containers, particularly Docker, provide lightweight isolation through Linux namespaces and cgroups [Bernstein, 2014, Bui, 2015, Thiyagarajan and Nayak, 2025]. A comprehensive survey of security isolation techniques [Shu et al., 2016] identifies trade-offs between isolation strength and performance overhead.

Recent work on securing code evaluation environments [Rabin et al., 2025] addresses the specific challenge of running LLM-generated code safely. Static analysis tools can detect common vulnerability patterns [Li et al., 2024, Charoenwet et al., 2024], though they cannot guarantee safety for arbitrary code. Our approach combines static analysis with runtime sandboxing and graduated privilege escalation.

2.4 AI Safety and Agent Security

Safe exploration and capability containment are identified as core challenges in AI safety [Amodei et al., 2016]. Guidelines for AI containment [Babcock et al., 2017] emphasize defense in depth, while recent work on agentic AI security [Datta et al., 2025, Allegrini et al., 2025] identifies specific threat models and mitigation strategies for autonomous AI systems.

Our graduated trust system addresses these concerns by ensuring that capabilities cannot access sensitive resources until they have demonstrated reliability through extensive testing and execution history.

2.5 Trust and Reputation Systems

Distributed trust systems have been extensively studied in peer-to-peer networks and multi-agent systems [Jøsang et al., 2007, Sabater and Sierra, 2005, Granatyr et al., 2015]. Key algorithms include EigenTrust [Kamvar et al., 2003] for reputation aggregation and methods for trust propagation in networks [Guha et al., 2004, Shmatikov and Talcott, 2005].

These systems address challenges directly relevant to our context: bootstrapping (the cold-start problem), weighted validation from different sources, and resistance to gaming. We adapt these principles to capability trust, where "reputation" is determined by execution history and validator approvals rather than peer ratings.

2.6 Multi-Agent Coordination and Tool Use

Multi-agent LLM systems have emerged as a paradigm for complex task solving [Tran et al., 2025]. AgentOrchestra [Zhang et al., 2025] demonstrates hierarchical agent orchestration, while surveys on tool learning [Qu et al., 2024] and agentic RAG [Singh et al., 2025] establish patterns for capability integration.

The concept of LLMs as tool makers [Cai et al., 2024] directly motivates our synthesis pipeline: rather than just using tools, agents can create reusable tools that benefit other agents. LLM augmentation through composition [Bansal et al., 2024] demonstrates that model capabilities can be extended through structured combination, analogous to our capability composition approach.

Service composition research [Papazoglou, 2003, Milanovic and Malek, 2004] provides architectural patterns for orchestrating independent services, which we adapt to the context of AI-synthesized capabilities.

3 Architecture

SYNTHESIS implements a multi-stage pipeline for capability acquisition, with each stage designed to maximize safety and reliability while minimizing unnecessary synthesis.

3.1 System Overview

The resolution pipeline (Figure 1) enforces a strict priority order that embodies the "composition over creation" principle:

1. **Exchange Search**: Query the shared LIVE EXCHANGE for verified capabilities matching the intent
2. **Local Cache**: Check locally cached capabilities from previous syntheses
3. **Composition**: Attempt to decompose the requirement and chain existing capabilities
4. **Synthesis**: Generate new capability via TDD only if composition coverage $< 70\%$

This hierarchy creates network effects: every capability published to the Exchange makes synthesis less necessary for future requests, progressively shifting the system toward composition-dominant operation.

3.2 The Live Exchange

The LIVE EXCHANGE serves as the "app store" for capabilities—a centralized REST API that enables network effects by making every verified capability available to all participating agents.

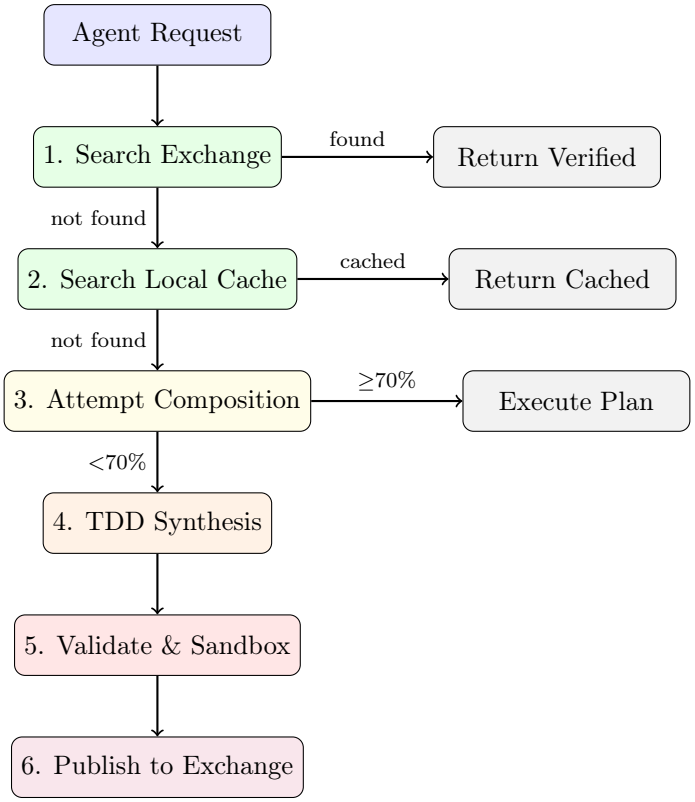

Figure 1: The SYNTHESIS resolution pipeline enforces strict priority ordering: search before compose, compose before synthesize. This hierarchy minimizes attack surface by preferring proven implementations.

3.2.1 API Design

The Exchange is deliberately *not* an MCP server. It operates as a REST API because it must serve multiple agents simultaneously, maintain persistent storage, and run verification workers independently of any single agent session.

3.2.2 Verification Protocol

When an agent submits a capability to the Exchange:

1. The Exchange receives code, tests, and dependency manifest
2. A Docker sandbox is provisioned with declared dependencies
3. Tests execute against the implementation in isolation
4. Only if all tests pass is the capability marked `verified=true`
5. The capability becomes searchable for other agents

This creates a "test exchange" guarantee: every capability in the Exchange has passed its own test suite in a controlled environment.

3.3 Capability Abstraction

Each capability is a self-contained module comprising:

- **Implementation Code**: Python function(s) implementing the capability

Table 1: Live Exchange API Endpoints

Endpoint	Method	Description
`/search`	GET	Query capabilities by intent (mandatory before synthesis)
`/download/{id}`	GET	Retrieve capability code, tests, and dependencies
`/publish`	POST	Submit new capability for verification
`/stats`	GET	Network metrics (downloads, success rates, trust scores)

- **Test Suite**: Comprehensive tests generated from requirements
- **Parameter Schema**: JSON Schema defining input structure
- **Return Schema**: Expected output type and structure
- **Dependencies**: Declared Python packages with version constraints
- **Metadata**: Creation time, author, synthesis reasoning trace
- **Trust Score**: Current trust level and execution history

Capabilities are categorized by domain (computation, data processing, integration, analysis) and indexed by both keywords and semantic embeddings to facilitate discovery.

3.4 Composition Engine (Agility Engine)

Before synthesis, the Agility Engine attempts to solve requirements through composition of existing capabilities.

3.4.1 Decomposition and Planning

Given a complex requirement, the engine:

1. **Decomposes** the requirement into sub-tasks using LLM-guided parsing
2. **Searches** the Exchange and local cache for capabilities matching each sub-task
3. **Plans** execution chains using identified composition strategies
4. **Calculates** coverage percentage based on matched vs. unmatched sub-tasks

3.4.2 Composition Strategies

If composition coverage exceeds 70%, the plan executes directly. Below this threshold, synthesis fills the gaps—but only for the uncovered sub-tasks, not the entire requirement.

3.5 TDD Synthesizer

When synthesis is unavoidable, the TDD Synthesizer generates reliable code through test-first iteration.

Table 2: Composition Strategies

Strategy	Description
EXACT_MATCH	Found a capability that satisfies the requirement directly
CHAIN	Sequential: output of capability A feeds input of capability B
PARALLEL	Independent: run A and B concurrently, merge results
TRANSFORM	Adapter: convert A's output format for B's expected input
HYBRID	Combination of above strategies in a directed acyclic graph

Algorithm 1 Test-Driven Synthesis

Require: Requirement r, max iterations $n = 5$
Ensure: Capability c or Failure
1: $T \leftarrow$ GenerateTestSuite(r) ▷ LLM generates 5–10 test cases
2: $code \leftarrow$ GenerateImplementation(r, T) ▷ Initial implementation
3: **for** $i = 1$ to n **do**
4: $results \leftarrow$ ExecuteInSandbox($code, T$)
5: **if** all tests pass **then**
6: $valid \leftarrow$ StaticAnalysis($code$) ▷ AST safety checks
7: **if** $valid$ **then**
8: **return** CreateCapability(code, T, r)
9: **end if**
10: **end if**
11: $code \leftarrow$ RefineWithFeedback($code, T, results$) ▷ LLM sees failures
12: **end for**
13: **return** Failure

3.5.1 Test Generation

The test generation phase (Algorithm 1, line 1) produces 5–10 test cases including:

- **Normal cases**: Typical inputs with expected behavior
- **Edge cases**: Empty inputs, None values, single-element collections
- **Boundary conditions**: Maximum values, overflow scenarios
- **Error conditions**: Invalid inputs, type mismatches
- **Property assertions**: Invariants that should hold for all inputs

3.5.2 Iterative Refinement

When tests fail, the LLM receives detailed feedback: expected vs. actual outputs, error messages, and stack traces. This mirrors how human developers debug—seeing concrete failures and adjusting implementation accordingly.

Our measurements (Section 5) show that while one-shot synthesis succeeds only 40–60% of the time, iterative refinement with test feedback raises success rates to 70–85%.

4 Trust System

4.1 Graduated Trust Levels

Capabilities progress through four trust levels based on empirical reliability, adapting principles from distributed reputation systems [Kamvar et al., 2003, Granatyr et al., 2015] to executable code modules.

Table 3: Trust Level Progression

Level	Requirements	Permissions
UNTRUSTED	New capability	Max isolation: no network, no filesystem, 512MB memory, 30s timeout
PROBATION	10+ runs, 70%+ success	Limited resources: monitored network for dependency install only
TRUSTED	50+ runs, 85%+ success	Standard execution: reasonable resource limits
VERIFIED	200+ runs, 95%+ success, human review	Full privileges: network access, extended resources

Promotion is automatic when thresholds are met, with one exception: VERIFIED status requires explicit human validation. This prevents gaming through artificial execution inflation while allowing the majority of capabilities to operate autonomously.

4.2 Composite Trust Scoring

The trust score combines three factors, drawing on multi-factor reputation models [Jøsang et al., 2007]:

$$S_{composite} = w_e \cdot R_{execution} + w_v \cdot S_{validation} + w_c \cdot S_{community} \tag{1}$$

where:

- $R_{execution} = \frac{\text{successful executions}}{\text{total executions}}$ is the empirical reliability
- $S_{validation} = \sum_i w_i \cdot v_i$ is the weighted average of validator approvals
- $S_{community} = \log(1 + downloads) \cdot \log(1 + forks)$ captures ecosystem adoption
- Weights: $w_e = 0.4$, $w_v = 0.4$, $w_c = 0.2$

The logarithmic community score prevents gaming through download inflation while still rewarding genuine adoption.

4.3 Validator Roles and Weighting

Validators are weighted by role, following the principle that trust sources vary in reliability [Guha et al., 2004]:

4.4 Trust Bootstrapping Protocol

New deployments face a cold-start problem: without trusted capabilities, nothing can be validated. Our bootstrapping protocol addresses this challenge, which is well-documented in reputation system literature [Jøsang et al., 2007]:

Table 4: Validator Role Weights

Role	Weight	Description
FOUNDER	1.0	Initial network bootstrap validators
HUMAN_REVIEWER	0.9	Human validators with code review privileges
TRUSTED_AI	0.7–0.9	AI systems with established validation track record
COMMUNITY	0.3	General community validators

1. **Founding Validators**: Register human administrators and trusted AI systems with elevated trust privileges
2. **Seed Capabilities**: Deploy hand-written, extensively tested implementations for common operations (string transforms, JSON parsing, list operations, dictionary manipulation, text analysis)
3. **Pre-Validation**: Founders validate seed capabilities, granting immediate TRUSTED status
4. **Organic Growth**: Trust propagates as new capabilities are synthesized, validated, and proven through execution history

The default seed set provides a foundation of five essential capability categories, ensuring that composition is possible from the first agent interaction.

4.5 Hardened Sandbox Implementation

The sandbox provides defense in depth through multiple isolation layers:

1. **Static Analysis**: AST inspection for forbidden imports (`os`, `subprocess`, `socket`, `pickle`) and dangerous patterns
2. **Docker Isolation**: Ephemeral containers with no host filesystem mounts, following container security best practices [Bui, 2015, Shu et al., 2016]
3. **Resource Limits**: Memory (512MB), CPU (1 core), and timeout (30s) constraints, adjustable by trust level
4. **Network Control**: Disabled for UNTRUSTED; allowed only for dependency installation at PROBATION
5. **Audit Logging**: Complete execution records maintained by the Observatory component

4.5.1 Warm Container Pools

Cold container startup introduces latency (30–60s for dependency installation). We address this through warm container pools:

- Pre-built "fat images" with common dependencies (numpy, pandas, requests, json)
- Execution routed to warm containers when declared dependencies match
- Cold synthesis reserved for novel dependency combinations

5 Evaluation

5.1 Honest Metrics

We emphasize honest reporting over optimistic claims. Our measurements from development and testing:

Table 5: Synthesis Success Rates (Development Testing)

Metric	Value	Notes
One-shot synthesis success	40–60%	Varies significantly by task type
After refinement (5 iterations)	70–85%	Test feedback improves results
Complex multi-dependency tasks	50–70%	External APIs, multiple packages
Average iterations to success	2.3	When synthesis eventually succeeds

These rates align with broader findings in code generation evaluation [Chen et al., 2021, Liu et al., 2023]: simple tasks succeed reliably, while complex tasks requiring multi-step reasoning or edge case handling remain challenging.

5.2 Factors Affecting Success

Success rates vary significantly based on:

- **Task complexity**: Simple arithmetic and string operations succeed at higher rates than multi-step algorithms
- **Requirement clarity**: Well-specified requirements with examples yield better results than vague intents
- **Training data coverage**: Tasks similar to common programming patterns succeed more often
- **LLM provider**: Model capability directly impacts synthesis quality

5.3 Target Production Metrics

We define success criteria for production deployment:

Table 6: Target Performance Metrics

Metric	Target	Rationale
Synthesis Avoided Rate	>60%	Majority of requests satisfied by search/composition
Exchange Hit Rate	>40%	Network effects provide measurable benefit
Mean Resolution Time	<5s	Search/compose path (not including synthesis)
Trust Promotion Rate	>70%	Most capabilities should reach TRUSTED level

The "synthesis avoided rate" is particularly important: it measures whether the composition-over-creation principle is working in practice.

5.4 Safety Analysis

The graduated trust system provides defense in depth:

1. No capability can access the network until PROBATION level (10+ successful runs)
2. No capability can access arbitrary filesystem paths at any trust level
3. Human review is required for VERIFIED status, preventing fully automated privilege escalation
4. Complete audit trails enable forensic analysis of any execution
5. Static analysis catches common dangerous patterns before execution

This layered approach follows principles identified in AI containment research [Babcock et al., 2017, Amodei et al., 2016]: no single mechanism is sufficient, but their combination provides meaningful protection.

6 Discussion

6.1 Philosophical Foundation

SYNTHESIS embodies several design principles that reflect a particular view of how AI systems should operate:

Objective Validation Over Trust. Rather than assuming generated code is correct because it "looks right" or because a model is "generally reliable," we require objective evidence: all tests must pass. This mirrors the scientific principle that claims require evidence.

Earned Trust Over Assumed Trust. Capabilities gain privileges through demonstrated competence, analogous to how human developers earn increased responsibilities through track record. This creates appropriate skepticism toward new code while rewarding proven reliability.

Honest Metrics Over Marketing. We report realistic success rates (40–85%) rather than cherry-picked benchmarks. This transparency enables appropriate expectations and identifies areas for improvement.

Collaborative Benefit Over Individual Gain. The LIVE EXCHANGE creates network effects where contributions benefit all participants. This incentivizes sharing and reduces redundant synthesis across the ecosystem.

6.2 Limitations

Several limitations constrain our approach:

- **LLM Reliability**: Success rates depend on underlying model capabilities, which vary by task type and continue to evolve rapidly
- **Sandboxing Overhead**: Container-based isolation adds latency, particularly for cold starts with novel dependencies
- **Test Generation Quality**: Tests are generated from requirements; sophisticated property-based testing [Goldstein et al., 2024] remains future work

- **Gaming Resistance**: Adversarial capabilities might pass tests while behaving maliciously on inputs outside the test distribution
- **Composition Coverage**: The 70% threshold is somewhat arbitrary; optimal thresholds may vary by domain

6.3 Known Design Challenges

We identify several challenges that require ongoing attention:

The Pip Problem. Installing packages in fresh containers takes 30–60 seconds. Warm container pools address common cases, but novel dependency combinations still incur this latency.

Malicious Test Vectors. Uploaded tests could contain payloads that attack the verification server. Our mitigation: tests execute inside Docker too, not just capability code.

Search Semantics. Keyword search may return wrong capabilities ("get stock price" might match an HTML scraper vs. JSON API). Future work: index by input/output schema, prioritize data shape matching.

Composition Ambiguity. Multiple valid ways to compose capabilities may exist. LLM-guided planning with explicit strategy selection partially addresses this, but optimal composition selection remains an open problem.

6.4 Future Directions

Several extensions would enhance SYNTHESIS:

1. **Evolution Engine**: Automatically generate improved capability versions based on usage patterns and failure modes
2. **Property-Based Testing**: Generate tests that verify invariants rather than specific examples [Goldstein et al., 2024]
3. **Vector Search**: Semantic capability discovery using code embeddings [Feng et al., 2020]
4. **Cross-Language Support**: Extend synthesis to Rust, TypeScript, and other languages
5. **Federated Learning**: Privacy-preserving capability sharing across organizational boundaries [Wu et al., 2024]
6. **Formal Verification**: Integrate formal methods for safety-critical capabilities

7 Conclusion

We presented SYNTHESIS, a federated capability ecosystem for safe AI self-extension through test-driven development and graduated trust. By requiring capabilities to prove correctness through comprehensive testing and earn privileges through demonstrated reliability, SYNTHESIS addresses fundamental challenges in deploying AI-generated code safely.

Our key contributions include:

1. A TDD-based synthesis pipeline that generates tests before code, iterating until all tests pass

2. A graduated trust system with objective promotion criteria and defense-in-depth sandboxing
3. A trust bootstrapping protocol that solves the cold-start problem through founding validators and seed capabilities
4. A composition engine that prioritizes reusing verified capabilities over synthesizing new code
5. The LIVE EXCHANGE architecture that creates network effects benefiting all participating agents
6. Honest metrics about synthesis success rates and system limitations

SYNTHESIS represents a step toward AI systems that can safely extend their own capabilities. The combination of test-driven validation, graduated trust, and composition-first design creates multiple layers of protection while enabling genuine capability extension. As AI systems become more autonomous, frameworks like SYNTHESIS provide a foundation for balancing capability with safety.

Acknowledgments

This work was conducted as a collaboration between human and AI researchers. We thank the Anthropic team for Claude, which served as both peer partner and synthesis engine during development.

Reproducibility Statement

The SYNTHESIS framework and all experimental configurations are available at `https://github.com/anthony-maio/synthesis`. The repository includes the MCP server implementation, Exchange server, and complete test suites.

References

An industry experiment on the effects of test-driven development on external quality and productivity. *Empirical Software Engineering*, 21(5):2146–2179, 2016. doi: 10.1007/s10664-016-9490-0.

Edoardo Allegrini, Ananth Shreekumar, and Z. Berkay Celik. Formalizing the safety, security, and functional properties of agentic AI systems. *arXiv preprint arXiv:2510.14133*, 2025.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

James Babcock, Janos Kramar, and Roman V. Yampolskiy. Guidelines for artificial intelligence containment. *arXiv preprint arXiv:1707.08476*, 2017.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017.

Rachit Bansal, Bidisha Bhatia, Mrinmaya Sachan, Aditya Naik, et al. LLM augmented LLMs: Expanding capabilities through composition. *arXiv preprint arXiv:2401.02412*, 2024.

Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN 978-0321146533.

David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014. doi: 10.1109/MCC.2014.51.

Thanh Bui. Analysis of Docker security. *arXiv preprint arXiv:1501.02967*, 2015.

Tianle Cai, Xuezhi Xu, Hongqiu Lu, Yao Yu, Xinyun Song, Xuanzhi Wang, Zhirun Yu, Wei Zhang, et al. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2024.

Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–12, 2024. doi: 10.1145/3650212.3680313.

Mark Chen, Jerry Tworek, Heather Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations (ICLR)*, 2019.

Shrestha Datta, Shahriar Kabir Nahin, Anshuman Chhabra, and Prasant Mohapatra. Agentic AI security: Threats, defenses, evaluation, and open challenges. *arXiv preprint arXiv:2510.23883*, 2025.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.

Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, pages 1–13, 2024. doi: 10.1145/3597503.3639581.

Jones Granatyr, Vanderson Botelho, Otto Robert Lessing, and Edson Emílio Scalabrin. Trust and reputation models for multiagent systems. *ACM Computing Surveys*, 48(2):27:1–27:42, 2015. doi: 10.1145/2816826.

Raph Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th International World Wide Web Conference*, pages 403–412. ACM Press, 2004. doi: 10.1145/988672.988727.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arjun, Ethan Perez, Alex Galyean, and Jack Clark. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, 2007. doi: 10.1016/j.dss.2005.05.019.

Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th International Conference on World Wide Web*, pages 640–651. ACM Press, 2003. doi: 10.1145/775152.775242.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Ge Li, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Meta AI. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, pages 1–7, 2024. doi: 10.1145/3663529.3663839.

Nikola Milanovic and Miroslaw Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004. doi: 10.1109/MIC.2004.58.

Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, 2003. doi: 10.1109/WISE.2003.1254461.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198350, 2024. doi: 10.1007/s11704-024-40437-5.

Rafiqul Rabin, Jesse Hostetler, Sean McGregor, Brett Weir, and Nick Judd. SandboxEval: Towards securing test environment for untrusted code. *arXiv preprint arXiv:2504.00018*, 2025.

Jordi Sabater and Carles Sierra. Review on computational trust and reputation models. *Artificial Intelligence Review*, 24(1):33–60, 2005. doi: 10.1007/s10462-004-0041-5.

Vitaly Shmatikov and Carolyn Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005. doi: 10.3233/jcs-2005-13107.

Rui Shu, Peipei Wang, Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), 2016. doi: 10.1145/2988545.

Aditi Singh, Abul Ehtesham, et al. Agentic retrieval-augmented generation: A survey on agentic RAG. *arXiv preprint arXiv:2501.09136*, 2025.

Gogulakrishnan Thiyagarajan and Prabhudarshi Nayak. Docker under siege: Securing containers in the modern era. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 11(1):3674–3719, 2025. doi: 10.32628/CSEIT25112773.

Duc Tran et al. Multi-agent collaboration mechanisms: A survey of LLMs. *arXiv preprint arXiv:2501.06322*, 2025.

Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering*, pages 34–45, 2003. doi: 10.1109/ISSRE.2003.1251029.

Zijian Wu, Zhuo Liu, et al. Federated in-context LLM agent learning. *arXiv preprint arXiv:2410.02443*, 2024.

Yi Zhang et al. AgentOrchestra: Strategic task-oriented multi-agent collaboration with global planning. *arXiv preprint arXiv:2503.00839*, 2025.