

Pusen Yi, Mallory Gilligan, Chloe Euston, Juliana Guarrera, Amelia Rogers

June 2nd, 2025

CS4992, Special Topics in Computer Science

Technical Credit Report

Introduction to Technical Credit:

In software engineering, the concept of technical debt is well established—it refers to shortcuts or compromises made during development that incur long-term costs in maintenance, scalability, and reliability. In contrast, the notion of technical credit has received far less attention. Technical credit represents deliberate engineering investments made with future system evolution in mind. These efforts, while not immediately necessary, are intended to reduce future complexity, support extensibility, and improve maintainability.

Technical credit often takes the form of architectural patterns such as abstraction layers, plugin systems, stateless design, or thorough documentation practices like architecture decision records (ADRs). Although these features may appear excessive or redundant in the short term, they frequently enable more efficient adaptation to evolving requirements.

Despite its relevance to software quality, there is currently no standardized method for identifying or measuring technical credit in codebases. This lack of tooling means that forward-thinking design decisions often go unnoticed or undervalued. This paper proposes a framework for detecting technical credit artifacts in software projects, combining heuristic-based analysis, prompt-guided inspection using large language models (LLMs), and a labeled dataset of positive and negative examples. By formalizing the identification of technical credit, we aim to support better recognition of long-term design quality and inform the development of automated analysis tools.

Foundations and Related Work:

Although the concept of technical credit is relatively new in name, it builds on decades of research into software quality, architectural decisions, and long-term maintainability. The closest analog is technical debt, introduced by Cunningham (1992), which has been extensively studied in terms of its identification, measurement, and mitigation strategies. Tools and metrics for

assessing debt—such as code smells, complexity metrics, and debt interest models—have helped formalize its role in software evolution. However, little parallel effort has been made to characterize or track deliberate, beneficial investments in software architecture.

Gorton, Bucaioni, and Pelliccione (2022) formally introduced the idea of technical credit as a counterpart to technical debt, advocating for its recognition in architectural evaluation. However, their work remains largely conceptual, and no frameworks or tools currently exist for detecting or tagging instances of technical credit in real-world systems.

This paper contributes to filling that gap by proposing a methodology to identify technical credit artifacts. Unlike prior work, which either critiques poor practices or documents best practices without detection mechanisms, we focus on formalizing and operationalizing the identification of positive design investments using a combination of heuristics, LLM-guided code inspection, and labeled examples.

Taxonomy of Technical Credit:

Technical credit manifests through several design strategies and decisions that improve a system’s adaptability, scalability, and maintainability. Technical credit promotes proactive engineering investments that down the line reduce development effort and risk. In this section, we will outline the common technical credit artefacts observed in software, when and why these are used, and highlight how each one can be systematically identified.

1. Circuit Breakers

- a. Use:** In distributed systems, circuit breakers are used to prevent repeated calls to services that are failing. Circuit breakers have a defined threshold and can monitor the number of failures. When this threshold is reached, the circuit “opens” and prevents any further calls to this failing service.
- b. Benefit:** By avoiding repeated failing calls, circuit breakers prevent cascading failure across system architecture and increase overall system stability. Circuit breakers are especially helpful when using systems with high service interdependencies, such as microservices.
- c. Identification:** Circuit breakers are usually implemented using libraries such as “pybreaker” in Python or custom wrappers around external service calls, such as

decorators. The code will also include state variables for the circuit (open or closed), failure thresholds, and timeouts.

2. Strategy Pattern

- a. Use:** The strategy pattern is used to define and encapsulate a family of behaviors. If there is one method that can be decided in several different ways, the strategy pattern is of good use (such as choosing moves, calculating price, sorting, estimations). This pattern consists of one strategy interface with a primary method, and as many implementations as needed. Since all strategies implement one strategy interface, they are interchangeable at runtime.
- b. Benefit:** In the strategy pattern, the interchangeability of strategies allows behaviors to differ independently from the context that uses it, which supports extensibility and reusability. It allows for new behaviors to be added without modifying existing code; all you need is to add a new implementation of the existing interface. This makes systems easier to evolve and adapt to changing requirements or desired behaviors.
- c. Identification:** The strategy pattern will always have one overarching interface, the strategy interface, usually with one or a few primary methods. There will be various implementations of this interface for each of the desired behaviors.

3. Adapter Pattern

- a. Use:** The adapter pattern allows for two incompatible interfaces to work together by introducing an intermediate “adapter” class. There is the target interface, which is the one the system expects, and the adaptee interface, which is usually an external interface or class that would otherwise not work in the system. The intermediate adapter class implements the target interface and holds the adaptee class as a field, and translates the adaptee class to work with the target interface.
- b. Benefit:** The adapter pattern is especially useful when integrating third-party libraries, refactoring old code, or joining different APIs under a single interface. This pattern reduces future cost and efforts by isolating the impact of external system changes within a single intermediate class, which improves maintainability and reuse.

- c. **Identification:** When the adapter pattern is used, there will always be a class implementing an expected interface that contains a field referencing an external system. The intermediate adapter class will oftentimes contain the word “adapter” in its name.

4. Model View Controller

- a. **Use:** The model view controller (MVC) pattern separates system design into three interconnected components: the model, view, and controller. The model contains data and logic, the view handles the user interface presentation, and the controller responds to user input by communicating between the model and the view.
- b. **Benefit:** The MVC structure improves modularity and maintainability. Since each component is separate, changes to one will not affect the other, which makes testing and debugging much simpler. The design also allows for passing a “read only” model into the view, so the view can display the model data but cannot alter it. This protects the model data from being changed through the UI, making the system more secure.
- c. **Identification:** In MVC, there should be clear, easy to identify separation of the model, view, and controller components. They will be separated into individual interfaces or separate directories.

5. Visitor Pattern

- a. **Use:** The visitor pattern separates behaviors from the objects that they act on. It allows you to define new methods without modifying the object classes themselves. Each behavior, or “visitor”, implements a visitor interface and contains methods to visit each object that it acts on. Each object class implements an accept method that takes a visitor as an argument and then calls the visitor’s corresponding visit method, passing itself as the input.
- b. **Benefit:** The visitor pattern enables extensibility, as it allows for adding new behaviors without changing the object classes being acted on. This preserves stability of object classes, and is particularly useful for systems expected to evolve various functionalities since you can easily add behaviors by simply implementing a new visitor class.

- c. **Identification:** With the use of visitors, there is almost always a visitor interface with “visit” methods for each object class. Within the object classes, there will typically be just one method, which is the “accept” method.

6. Iterator

- a. **Use:** The iterator pattern provides a standardized way to access and traverse elements in a collection without exposing its internal structure (lists, stacks, trees, etc). It does so by defining a separate object, called an iterator, that manages the traversal. The iterator accesses elements one at a time, typically through methods like “next” and “iter” in Python.
- b. **Benefit:** The iterator pattern separates data storage from its processing, improving encapsulation and flexibility. This makes it easy to change the structure of the data without affecting the code that uses it. Overall, iterators offer a reusable and consistent way to interact with data of all types.
- c. **Identification:** When iterators are used in Python, iterator classes will implement iteration methods such as “next” and “iter”.

7. Platform Abstraction Layers

- a. **Use:** Platform abstraction layers create flexible interfaces for interacting with external systems. They define a family of related objects behind abstraction. This allows an application to use them without depending on their concrete implementations. This makes it easier to change or swap implementations without changing the system’s core logic and structure.
- b. **Benefit:** Abstraction layers improve a system’s adaptability. They prevent future rewrites of code with functionality changes, as you can change their implementations without affecting the overall architecture. They also improve testability, since abstraction layers can be mocked easily.
- c. **Identification:** Abstraction layers are identified easily, as they will be labeled as abstract in the naming of the class. They will have one or many concrete implementations.

8. Proxy

- a. **Use:** The proxy pattern is used to provide a placeholder or substitute for another object in order to manage access to that object. A proxy can add additional

behavior, such as access control or lazy initialization, without changing the object being accessed. It essentially acts as a stand in interface that can wrap extra logic in order to access a given object.

- b. Benefit:** A proxy separates extra tasks into a separate component. Proxy's make it easier to add features, such as security checks and caching, without adjusting the primary code. The original object will hold just its original functionality, while the proxy adds the extra desired steps. This makes the system easier to manage and update.
- c. Identification:** Proxies will implement the same interface as the target class and will hold that class as a field (where the target class is the class being accessed). Within methods, the proxy performs extra logic before delegating calls to the target class. This extra logic may also appear as wrapper classes or decorators.

9. Chain of Responsibility

- a. Use:** The chain of responsibility pattern that lets you pass requests along a chain of handlers. Each handler either processes the request itself, or passes it along in the chain. This is most often used when multiple objects might handle a request, but the specific handler is not known ahead of time.
- b. Benefit:** Chain of responsibility improves system flexibility by decoupling senders and receivers of requests. The code sending the request does not have to be changed every time a new handler is added or removed; if a new handler is needed, simply create it and add it to the chain.
- c. Identification:** When chain of responsibility is used, the handler classes will all implement the same interface. Each handler holds a reference to the next handler in the chain. In Python, this pattern often appears in pipelines, middleware stacks, or event processing systems.

10. Template Method

- a. Use:** The template method defines a skeleton of an algorithm in a base class that allows for subclasses to override and fill in certain steps. The overall process structure stays consistent, but allows for some variety in individual steps.

- b. Benefit:** This pattern centralizes shared behavior into a single template, which promotes adaptability and consistency. New processes can be implemented easily, and duplicate code can be avoided by only changing the necessary steps.
- c. Identification:** Using the template method involves a base class with a concrete method that calls one or many abstract methods intended to be implemented by subclasses. The base class provides default behavior or leaves it for subclasses to define.

The artefacts above demonstrate the ways in which technical credit manifests through thoughtful design choices that promote a system's long-term usability. Each design choice involves upfront thinking and work that makes software easier to adapt, scale, and maintain in the future. Recognizing and identifying these artefacts in software can give a better evaluation of a system's architectural foresight and ensure the use of engineering practices that invest in future efficiency and stability.

Methodology: Operationalizing Technical Credit Detection:

The operationalization strategy effectively bridges the gap between theoretical maintainability principles and actionable code review practices by establishing concrete measurable criteria that reviewers can consistently apply. The focus on observable design patterns, interface-implementation separation, dependency injection, and comprehensive architectural documentation, provides a robust foundation because these elements directly correlate with a system's capacity for evolution and adaptation. This approach transforms subjective quality assessments into systematic evaluations, enabling both human reviewers and automated tools to identify technical credit with greater precision and consistency across diverse development contexts.

The scalability of this framework depends critically on establishing clear thresholds and validation mechanisms that can accommodate domain-specific variations while maintaining analytical rigor. Static analysis of structural patterns offers efficiency and reproducibility but the true measure of technical credit lies in how these design choices facilitate actual maintenance and extension activities over time. Integrating longitudinal metrics—such as change velocity, defect rates, and feature development cycles—with structural assessments would strengthen the correlation between observable code characteristics and genuine maintainability outcomes,

ultimately providing empirical validation for the heuristic framework's effectiveness in predicting long-term software sustainability.

LLM Assisted Analysis:

In this phase, we used large language models (LLMs) to support our analysis of technical credit in codebases. By crafting targeted prompts, we guided the model to inspect and interpret code snippets, documentation, and architectural patterns that may indicate deliberate long-term design decisions. The LLM helped flag potential indicators such as abstraction layers, plugin systems, and well-documented architecture decisions, which we then verified against our practical criteria. While the model's output required human review to confirm accuracy, it significantly accelerated our ability to surface and examine candidates for technical credit. The approach highlighted the potential for integrating LLMs into semi-automated tooling for architectural analysis.

Results and Interpretation:

As a method of finding and identifying technical credits in software, we created a Retrieval-Augmented Generation (RAG) System that communicates between two databases and an LLM. The first database is a technical credit description database, which contains articles, documentation, and definitions describing various forms of technical credit. The second database is a vector database, which is built from our training data on technical credit. Our training data consists of simple repositories for various types of technical credit, including all of the patterns which we defined in the "Taxonomy of Technical Credit" section. Upon user input of a repository or segment of code, the RAG system retrieves relevant entries from both databases and feeds them into the LLM as contextual input. When retrieving relevant code from the vector database a similarity search is performed. This finds the most relevant code and hopefully does not confuse the LLM with unnecessary context. The LLM then analyzes the input, identifies instances of technical credit, and generates a structured response summarizing the findings. Through testing, we found that our RAG system reliably identifies the most prominent technical credit artefacts in repositories, with Claude and Gemini as LLMs showing the highest relevance in outputs.

The following example demonstrates the effectiveness of our implementation. We selected the public repository TicTacToe by praypratyay due to its straightforward structure, real-world application, and incorporation of technical credits. Prior to analysis, we identified a set of technical credits we expected to find within the codebase. In this section, we compare the outputs of various LLM model combinations against these expectations to evaluate their accuracy and alignment.

```
[26]: response2 = graph.invoke(anthropic_and_chroma_state({
    "question": "What tech credits can you identify from this repo?",
    "url": "https://github.com/praypratyay/TicTacToe",
    "branch": "main"}))
print("----Response from Anthropic with gemini embeddings----")
print(response2["answer"])

----Response from Anthropic with gemini embeddings----
Based on the provided code snippets, I can identify several technical credits:

**1. Strategy Pattern Implementation**
The code demonstrates a clear Strategy pattern with 'BotPlayingStrategy' as the abstract base class and concrete implementations ('EASYBotPlayingStrategy', 'MEDIUMBotPlayingStrategy', 'HARDBotPlayingStrategy'). This creates technical credit by making bot difficulty algorithms interchangeable and allowing easy addition of new difficulty levels without modifying existing code.

**2. Factory Pattern for Strategy Creation**
The 'BotPlayingStrategyFactory' class provides a centralized way to create different bot playing strategies. This abstraction layer creates technical credit by decoupling strategy creation from client code, making it easier to modify strategy instantiation logic in the future.

**3. Encapsulation with Property Pattern**
The 'Board' class uses private attributes ('__size', '__grid') with proper encapsulation, as mentioned in the comments about using getters/setters for validation and controlled access. This creates technical credit by providing a clean interface that can be extended with validation logic while maintaining backward compatibility.
```

Figure 1: Claude for analyzation & Google Gemini for embedding

Through testing we concluded that this combination is the best in terms of accuracy and reasoning. Strategy, Factory and Property patterns are all present in the selected code and the reasoning is accurate in describing the rationale.

```
[30]: response1 = graph.invoke(openai_and_chroma_state({
    "question": "What tech credits can you identify from this repo?",
    "url": "https://github.com/praypratyay/TicTacToe",
    "branch": "main"}))
print("----Response from Openai with gemini embeddings")
print(response1["answer"])

----Response from Openai with gemini embeddings
The repo demonstrates the use of the Strategy Pattern as technical credit. The BotPlayingStrategy classes (EASY, MEDIUM, HARD) encapsulate different algorithms for making moves, allowing them to be interchangeable independently from the Board class or clients using them. The BotPlayingStrategyFactory provides simple selection of strategies based on difficulty, adhering to the pattern's intent of varying algorithms without changing clients.

Additionally, the BotPlayingStrategy base class uses abstract methods, and the subclasses override behavior, reflecting aspects of the Template Method pattern by defining the skeleton of the algorithm and allowing subclasses to customize steps. The Board class uses property decorators and private attributes to encapsulate internal state, demonstrating good design decisions that enable ease of future modifications, which aligns with the idea of accruing technical credit by easing cost-effective future changes.

Overall, key tech credits are the Strategy Pattern (flexible algorithm encapsulation and interchangeability) and Template Method pattern (defining algorithm skeleton with overridable steps).
```

Figure 2: OpenAI for analyzation & Google Gemini for embedding

The main difference between this response and the previous is the length and justification. We can see that only one main type of technical credit was suggested in the strategy pattern. While this model captured accuracy, it was outperformed by Claude's analyzation of multiple patterns.

```
[27]: response3 = graph.invoke(anthropic_and_hf_state({
    "question": "What tech credits can you identify from this repo?",
    "url": "https://github.com/praypratyay/TicTacToe",
    "branch": "main"}))
print("---Response from Anthropic with hugging face embeddings---")
print(response3["answer"])

---Response from Anthropic with hugging face embeddings---
Based on the provided code snippets, I can identify the following technical credits:

**Strategy Pattern Implementation**: The code demonstrates a clear implementation of the Strategy pattern in the `BotPlayingStrategy` class. The abstract `BotPlayingStrategy` base class defines the interface, while concrete implementations (`EASYBotPlayingStrategy`, `MEDIUMBotPlayingStrategy`, `HARDBotPlayingStrategy`) provide different algorithms for bot moves. This design allows the game to switch between different bot difficulty levels without modifying client code.

**Builder Pattern Setup**: The `GameBuilder` class shows the beginning of a Builder pattern implementation with its fluent interface methods (`setDimension`, `setPlayers`, `setwinningStrategies`). This pattern enables flexible game configuration by allowing clients to construct game objects step-by-step with different combinations of parameters.

**Encapsulation and Property Management**: The `Board` class demonstrates good encapsulation practices by using private attributes (`__size`, `__grid`) and mentions the use of property decorators for controlled access. This approach provides validation capabilities and protects internal state from direct external modification, making future changes to the board implementation easier to manage.
```

Figure 3: Claude for analyzation & HuggingFace for embedding

Here we see a new technical credit returned. The builder pattern is used here but is certainly not one of the main technical credits we identified. Here we can see the interesting results when different embedding models are used.

```
[28]: response4 = graph.invoke(openai_and_hf_state({
    "question": "What tech credits can you identify from this repo?",
    "url": "https://github.com/praypratyay/TicTacToe",
    "branch": "main"}))
print("---Response from Openai with hugging face embeddings---")
print(response4["answer"])

---Response from Openai with hugging face embeddings---
The repo demonstrates the use of the Strategy Pattern as a technical credit. This is evident from the BotPlayingStrategy interface with multiple concrete strategies (EASYBotPlayingStrategy, MEDIUMBotPlayingStrategy, HARDBotPlayingStrategy) encapsulating different algorithms for making moves. Additionally, the code allows interchangeability of these strategies independently from the Board or game logic, aligning with standard definitions of the Strategy Pattern.
```

Figure 4: OpenAI for analyzation & HuggingFace for embedding

As you can see this response returns the least amount of information. We have come to the conclusion that this combination returns the worst results. While the result of the strategy pattern is correct, this model was only able to identify one case of technical credit while the other models all identified at least 2.

```
[29]: response5 = graph.invoke(anthropic_and_chroma_state({
    "question": "Does this repo have Circuit Breaker as a tech credit",
    "url": "https://github.com/praypratyay/TicTacToe",
    "branch": "main"}))
print("---Response from Anthropic with gemini embeddings---")
print(response5["answer"])

---Response from Anthropic with gemini embeddings---
Based on the provided code snippets and documentation, I cannot identify any Circuit Breaker implementation in this repository. The code shows implementations of the Strategy pattern and Template Method pattern for a bot playing strategy system, but there are no safeguards against cascading failures or mechanisms to handle transient performance degradation. According to the documentation, Circuit Breakers are technical credit because they guard against cascading failures and ensure system resilience, but no such protective mechanisms are present in the examined code.
```

Figure 5: Direct Question to Claude + Google Gemini combination

In the above example we asked a direct question about a certain technical credit. In this case the LLM responded correctly. This repo does not exemplify a Circuit Breaker pattern. In the beginning of our testing process we had an issue where the LLM was consistently returning

circuit breaker no matter the code inputted. In this case responding that there is no circuit breaker pattern is an indicator of a well functioning project.

When testing outside of the RAG the biggest issue is that most chatbots are not aware of what tech credit is. Most often it gives a technical overview of the project.

Conclusions and Future Work:

This project works to build a foundation for formalizing the identification of Technical Credit in software systems by developing a methodology that combines structured heuristics and LMM-guided analysis. Our approach underscores the potential for developing tools that recognize and encourage sustainable software design by building a practical framework for identifying key architectural investments that can be systematically surfaced and evaluated.

However, the scope of this project was constrained by several limitations. We worked with a relatively small data set that contains biases towards well-known, generalized patterns. Additionally, false positives also pose a risk to the validity of our detection results, potentially inflating the observed frequency of Technical Credit in a dataset. Future work may address these limitations by expanding the dataset across more diverse domains and project types to improve the generalizability of TC. Moreover, we envision a fully integrated RAG system, fine-tuned on a large, curated corpora of data, with higher precision. Enhancing the testing and evaluation of results would also allow for a more empirical validation of TC's long-term value. With broader, more robust fine-tuning elements, the precision of TC identifications could become more improved and reliable. As our current system uses a single vector database for TC classifications, future iterations would benefit from incorporating a MoE (Mixture of Experts) machine learning technique. Rather than relying on one monolithic model, MoE could divide specialized services to handle different styles of software artifacts. Embedding MoE into the RAG pipeline would offer more targeted recommendations on design investments at different stages of software development. Over time, this could evolve into a system that not only identifies TC artefacts but a more intelligent "Technical Credit Advisor": an apparatus that is not only evaluative but also prescriptive, guiding developers during code reviews, refactorings, and architectural revisions. These recommendations could improve to become project specific, based on evolving systems and projected trajectories. Giving agency to these conceptual methodologies lays the groundwork

for a future where software architecture is evaluated not solely by present utility, but by its readiness for change.