The Name of the Title Is Hope

BEN TROVATO* and G.K.M. TOBIN*, Institute for Clarity in Documentation, USA
LARS THØRVÄLD, The Thørväld Group, Iceland
VALERIE BÉRANGER, Inria Paris-Rocquencourt, France
APARNA PATEL, Rajiv Gandhi University, India
HUIFEN CHAN, Tsinghua University, China
CHARLES PALMER, Palmer Research Laboratories, USA
JOHN SMITH, The Thørväld Group, Iceland
JULIUS P. KUMQUAT, The Kumquat Consortium, USA

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Do, Not, Use, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Ben Trovato, G.K.M. Tobin, Lars Thørväld, Valerie Béranger, Aparna Patel, Huifen Chan, Charles Palmer, John Smith, and Julius P. Kumquat. 2018. The Name of the Title Is Hope. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

ACM's consolidated article template, introduced in 2017, provides a consistent LaTeX style for use across ACM publications, and incorporates accessibility and metadata-extraction functionality necessary for future Digital Library endeavors. Numerous ACM and SIG-specific LaTeX templates have been examined, and their unique features incorporated into this single new template.

*Both authors contributed equally to this research.

Authors' Contact Information: Ben Trovato, trovato@corporation.com; G.K.M. Tobin, webmaster@marysville-ohio.com, Institute for Clarity in Documentation, Dublin, Ohio, USA; Lars Thørväld, The Thørväld Group, Hekla, Iceland, larst@affiliation.org; Valerie Béranger, Inria Paris-Rocquencourt, Rocquencourt, France; Aparna Patel, Rajiv Gandhi University, Doimukh, Arunachal Pradesh, India; Huifen Chan, Tsinghua University, Haidian Qu, Beijing Shi, China; Charles Palmer, Palmer Research Laboratories, San Antonio, Texas, USA, cpalmer@prl.com; John Smith, The Thørväld Group, Hekla, Iceland, jsmith@affiliation.org; Julius P. Kumquat, The Kumquat Consortium, New York, USA, jpkumquat@consortium.net.

If you are new to publishing with ACM, this document is a valuable guide to the process of preparing your work for publication. If you have published with ACM before, this document provides insight and instruction into more recent changes to the article template.

The "`acmart`" document class can be used to prepare articles for any ACM publication — conference or journal, and for any stage of publication, from review to final "camera-ready" copy, to the author's own version, with *very* few changes to the source.

2 Template Overview

As noted in the introduction, the "`acmart`" document class can be used to prepare many different kinds of documentation — a double-anonymous initial submission of a full-length technical paper, a two-page SIGGRAPH Emerging Technologies abstract, a "camera-ready" journal article, a SIGCHI Extended Abstract, and more — all by selecting the appropriate *template style* and *template parameters*.

This document will explain the major features of the document class. For further information, the *LaTeX User's Guide* is available from https://www.acm.org/publications/proceedings-template.

2.1 Template Styles

The primary parameter given to the "`acmart`" document class is the *template style* which corresponds to the kind of publication or SIG publishing the work. This parameter is enclosed in square brackets and is a part of the `documentclass` command:

```
\documentclass[STYLE]{acmart}
```

Journals use one of three template styles. All but three ACM journals use the `acmsmall` template style:

- `acmsmall`: The default journal template style.
- `acmlarge`: Used by JOCCH and TAP.
- `acmtog`: Used by TOG.

The majority of conference proceedings documentation will use the `acmconf` template style.

- `sigconf`: The default proceedings template style.
- `sigchi`: Used for SIGCHI conference articles.
- `sigplan`: Used for SIGPLAN conference articles.

2.2 Template Parameters

In addition to specifying the *template style* to be used in formatting your work, there are a number of *template parameters* which modify some part of the applied template style. A complete list of these parameters can be found in the *LaTeX User's Guide.*

Frequently-used parameters, or combinations of parameters, include:

- `anonymous,review`: Suitable for a "double-anonymous" conference submission. Anonymizes the work and includes line numbers. Use with the `\acmSubmissionID` command to print the submission's unique ID on each page of the work.
- `authorversion`: Produces a version of the work suitable for posting by the author.
- `screen`: Produces colored hyperlinks.

This document uses the following string as the first command in the source file:

```
\documentclass[manuscript,screen,review]{acmart}
```

3 Methodology

To identify and evaluate common refactorings in LangChain and LangGraph projects, we performed a series of pointed queries against the GitHub REST API. Queries were directed at the search/commits endpoint using a custom python script authenticated with a GitHub API token. The commit mining script is designed such that queries are made with a particular combination of keywords and a maximum number of commits to return to prevent uncontrolled data acquisition.

To prevent duplication, logic is implemented to omit commits already present in the dataset. In the event the API returned a commit meeting the specified requirements, the SHA, URL, repository, and committer username were appended to the CSV file. Every commit on GitHub has a unique SHA, so this was used as the primary key to differentiate between distinct observations. The commit repository and author username were both derived directly from the URL.

Each query returned commits with a series of keywords present in the commit message. This ensured relevant data retrieval given the commit author explicitly stated a LangChain or LangGraph refactoring as their intention. The term "refactoring" was present in each query along with either "LangChain" or "LangGraph" and an associative keyword such as "maintainability", "cost", "latency", or "error-handling". Our terms of interest were compiled after a rigorous review of LangChain documentation and exploratory analysis of LangChain-related commits. This approach allowed our team to filter to a subset of commits with a much higher likelihood of relevance.

The addition of a third keyword of interest also ensured a more diverse distribution across different contributors to technical debt such that a variety of technical debt categories can be well-represented in the data. We did not require any further filtering by commit date for relevance because the LangChain library was released in October of 2022. Thus, the presence of the LangChain dependency satisfied any considerations for relevance in our study.

Refactorings that may have perceived benefits to technical debt but do not directly modify the implementation of LangChain or LangGraph components were considered outside of the scope of this study. While such refactorings may improve agentic software applications they are typically context-dependent and may not be broadly applicable to LangChain implementations. By narrowing our scope to consider only direct LangChain and LangGraph code, refactorings can be generalized such that developers can follow these practices in any application built using the LangChain or LangGraph libraries.

During the manual examination process, phrases or claims present in the commit message were used to aid in intent classification. After a review of existing patterns in our dataset, we identified seven technical debt categories: cost, performance, reliability, maintainability, scalability, compatibility, and understandability. Cost refers to the monetary efficiency of the system. The vast majority of projects evaluated call on external language model providers via an API. The owner of this API token is then charged for each query and typically for the number of tokens processed. Thus, cost as technical debt is paramount because administrators of the software can lower the monetary investment associated with running the application. Typically cost can be lowered by reducing the number of input tokens present in queries or by reducing the number of queries made when possible. Performance refers to improving the perceived latency of an application. Technical debt associated with performance can negatively impact the user experience of an application. Reliability refers to the system's ability to mitigate failures or undesired behavior in the application. In LangChain and LangGraph applications this can include graceful exception-handling, proper retry logic when calling external APIs, or simply producing the desired response or action. Maintainability is a technical debt category that encompasses a developer's ability to maintain or expand upon the existing application during iterative development. Scalability in this project's scope is defined as the ability of an application to scale with an increase in users or data. Applications should be designed such that they can scale without significant reductions to other debt categories such as performance and reliability. Compatibility refers to the ability of an application to integrate with different infrastructure or platforms. Agentic applications built using LangChain oftentimes require external database technologies or cloud providers. Properly designed projects should be able to seamlessly migrate between providers of such technologies. Understandability is defined as the comprehensibility of the code base itself. Issues associated with understandability can hinder the development process as members of a team may not be able to effectively read and understand existing contributions from other members of the team. Each confirmed refactoring contributes to one or more of these technical debt categories.

Upon the retrieval of relevant commits, we then required a vigorous process of manual evaluation for each commit to determine whether or not the commit contains a valid refactoring and to assign it to the proper technical debt category. This process included reading commit messages to deduce developer motivation and manually parsing through thousands of lines of code to confirm relevant changes to the code base that reduce technical debt while maintaining functionality.

Static analysis of commit diffs allowed for the omission of false positives where a commit author claimed a particular refactoring but did not yet implement such a change in the code base. Additionally, some of the commits retrieved contained the keyword "langchain" or "langgraph" but did not explicitly leverage the official langchain or langgraph libraries. In such cases, it was determined that developers were referring to langchain or langgraph patterns in their code. That is to say that while they did not use the official langchain library, they followed a similar architecture in their implementation.

Analysis of commit diffs is a meticulous process that requires familiarity LangChain abstractions and interfacing with language model APIs. Accordingly, one of our authors is a professional software engineer specializing in building agentic applications using the LangChain and LangGraph libraries. Another author is an experienced researcher specializing in refactoring and technical debt in software engineering projects.

Generally, developer intentions were derived from the commit message while the commit diff was used to validate tangible changes associated with the motivation. Pull requests were also evaluated to further explain developer intention when available. Developers frequently claim measurable latency or cost reductions in their commit messages or pull requests. These claims are difficult to validate in isolation. Instead, this study outlines recurring patterns across a diverse sample of public repositories on GitHub.

4 Results

4.1 Quantitative Analysis

The study includes 500 commits from 328 distinct open-source projects publicly available on GitHub. Of the 500 commits evaluated, our team extracted 169 confirmed refactorings representing 33.8% of the dataset. The remaining 331 (66.2%) contained keywords of interest but upon manual analysis our authors determined it either did not contain a true refactoring or contained a refactoring not associated with the LangChain and LangGraph libraries. These confirmed refactorings were then classified into 30 different distinct refactorings which we deduced from LangChain expertise and pattern recognition. Within the true positive subset, commits contain a mean of 1.73 distinct refactorings, indicating that developers frequently implement multiple refactorings in tandem. Each refactoring is also assigned to one or more technical debt categories which can be seen in Table 2. True refactoring commits also addressed a mean of 1.6 distinct technical debt categories. Our team compiled these refactorings such that developers leveraging the LangChain and LangGraph libraries can make these changes to enhance their code and alleviate technical debt. Given that our data is well distributed across different repositories we can validate the refactorings extracted because they occur independently across many projects. In isolation, a self-admitted refactoring is admissible. This is why our team required an empirical study to deduce repeated patterns found in the open-source community. Several refactorings that occurred once in isolation were omitted from this study as they were deemed anecdotal.

n) Message Handling Refactor. The most common refactor among validated refactorings was the Message Handling refactoring, occurring independently 40 times in 23.5% of true positive commits. This refactor alters message processing to handle the metadata associated with LangChain message objects. Developers may bolster their implementations by using LangChain's message objects directly without any proprietary dictionaries or by using the metadata returned by certain message objects for additional context. This refactoring in turn enhances the maintainability, reliability, and understandability of a project. A frequent error made by developers is to naively evaluate the messages returned by turns without accounting for potentially unexpected results, such as tool call data. The Message Handling refactoring typically occurred in isolation, only occurring with another distinct refactor in 9 commits representing 22.5% of Message Handling occurrences. This behavior was unique among refactorings, as others generally occurred in large overhauls of the existing source code with many distinct changes.

n) Alter State Schema to Use Add Messages Reducer. The second most frequently repeated refactoring was the implementation of LangGraph's add messages reducer to the state schema constructor, occurring in 13% of validated commits. Though this refactoring also frequently co-occurs with other refactorings to graph state schema. The introduction of the add messages reducer co-occurs with other graph state expansions in 18 percent of it's instances in our dataset. The add messages reducer is also frequently introduced in the same commit as the division of graphs into sub-agents, presumably because the reducer allows for concurrent message updates to state which is advantageous when multiple agents are working simultaneously on independent tasks. Such a co-occurrence is present in 27% of the commits including the add messages reducer refactoring. These results indicate a possible codependent relationship in which each refactoring benefits from the implementation of the other. Both the sub-agents and add messages reducer refactorings address the scalability of an application, so developers may commit both refactorings with the singular goal of enhancing scalability issues.

n) LangGraph-Specific Refactorings. Each refactoring is listed along with their frequency in the data and associated technical debts in Table 2. The discovered refactoring types consist of 8 (28.5%) LangGraph-specific changes and 20 (71.4%) changes that can be applied to both LangGraph and LangChain implementations. There are no alterations that apply exclusively to LangChain projects because LangGraph applications leverage primitive LangChain objects such as message types, models, or retrievers. Regardless, it is still important that we distinguish the number of true positive commits that are composed of graph or chain implementations, as these fundamentally differ in their agent orchestration. Of the 170 true positive commits, 92 (54.1%) were graph implementations using LangGraph while the remaining 78 (45.9%) observations were chain or sequence implementations. Despite an equal number of queries made containing each framework and LangGraph's relative immaturity, we observed more valid refactorings in graph implementations.

Of the LangGraph specific refactorings, the add messages reducer refactor and division into subagents accounted for 43.5% of the LangGraph specific refactorings appearing in 24 and 16 distinct observations respectively.

n) Performance Enhancing Refactorings. Of the refactoring types discussed, 9 (41%) contribute to enhancing performance. Making performance the most frequently addressed technical debt in distinct refactorings. Indicating developers have different means by which to address latency in langchain applications. Additionally, performance is addressed in 52 unique commits representing 31% of the true positive commits in our dataset. This makes performance the third most frequent technical debt category addressed. Of the distinct refactorings discovered, the two most commonly addressed are the Add Messages Reducer and Asynchronous Invocation. The Add Messages Reducer refactor enhances the performance of graphs by allowing for concurrent updates to state. The Asynchronous Invocation refactor has similar motivation by allowing for graphs or chains to be executed asynchronously, preventing blocking and thereby reducing latency. For instance, in a synchronous implementation when an API call is made to an external LLM provider, all other processes must wait for this action to complete before resuming execution, thus creating reducible latency.

n) Cost Enhancing Refactorings. Within the true positive dataset, cost is addressed in 30 distinct commits representing 17.6% of true refactors, making Cost the fifth most frequently addressed technical debt category. The Cache Recent Query Responses, Model Initialization Arguments, Context Trimming or Summarization, and Specialized Model Routing refactorings each contribute to mitigating the technical debt associated with Cost. This represents 18% of the discovered refactoring types. The Cache Recent Query Responses refactor contributes to mitigating cost by preventing unnecessary LLM calls on repeat queries thereby reducing the number of queries made to LLMs when possible. Context Trimming or Summarization can significantly reduce cost, particularly in multi-turn interactions, by reducing the number of tokens with which an LLM is called. If a language model is called with all messages from previous turns on a particular thread this causes an uncontrolled increase in the number of tokens present in each query; this is the behavior that the Context Trimming and Summarization refactor prevents. Certain model initialization arguments such as maximum iterations, maximum tokens, or maximum retries can act as guardrails preventing costly agent behavior. By setting tuning these parameters correctly in the context of the application, developers can prevent users from making calls with many tokens or prevent agents from making many LLM reasoning calls on a single user request. Thus, this refactoring type can greatly enhance the cost efficiency within an application while keeping the observable behavior relatively unaltered. The Specialized Model Routing Refactor can significantly reduce the cost of an application by routing to cheaper LLM APIs when the query is simple and only calling expensive state of the art models when a query is complex or intensive. The pricing of LLM APIs varies greatly depending upon provider, model size in parameters, and recency of the model release. As a result, routing to different models based on the perceived query difficulty is an effective way to reduce cost while maintaining the quality of agent actions.

n) Maintainability Enhancing Refactorings. Maintainability is the second most frequent technical debt category addressed within the true refactoring subset. 85 commits (50%) addressed maintainability as technical debt. The refactoring types that contribute to enhancing maintainability include: ChatPromptTemplate, Message Handling, Generic Model Interface, Model Initialization Arguments, Migrate to Core Modules, Initialize tool with @tool decorator, Modularization of Components, and Graph Restructure. This technical debt category has the greatest coverage across refactoring types with 10 (45.5%) types contributing to Maintainability. The ChatPromptTemplate refactor enhances maintainability by providing one consistent and reusable prompt template throughout the entirety of a langchain or langgraph implementation. This allows for the seamless prepending of a system prompt when necessary and the prompt template integrates well with langchain model objects. The Message Handling refactor contributes to maintainability by improving the processing of new message objects and their associated metadata. When developers make changes that leverage LangChain's native message objects this prevents the need for manually appending and parsing a dictionary of role keys and content values. Commits leveraging the Generic Model Interface refactor type enhance maintainability by allowing developers to seamlessly switch between models by changing a single value within the model object. If a developer uses a provider-specific model object then opting to a different model or provider will require alterations to the model arguments and the objects with which the model interacts. These changes can quickly become expansive throughout the code base. By using a universally compatible model object we can prevent unnecessary changes when switching between models and improve the maintainability of an application. The Initialize Tools with @tool Decorator refactor contributes to maintainability by defining tools and their functionality simultaneously within the code base. The tool decorator does so by packaging the tool function as a LangChain tool object and using the function's docstring as the tool description to be provided to an agent to improve tool selection. The alternative would be to define the tool object and tool functionality separately which may make it difficult for developers to maintain the application as they may need to make changes in two distinct locations for a single tool. The modularization of components refactor contributes solely to maintainability as technical debt. In this refactor developers split distinct langchain or langgraph components into separate files which allows them to easily locate components within the file tree of an application.

n) Reliability Enhancing Refactorings. Reliability was the most frequently addressed technical debt category in the true refactorings subset. 90 of the 170 observed refactorings contributed to reliability as technical debt. Of the 22 distinct refactoring types, 8 contribute to improving reliability. These include: ChatPromptTemplate, Message Handling, Model Initialization Arguments, Compile with Checkpointer, Structured Tool Wrapper, @tool decorator, model fallbacks, and specialized model routing. The Model Initialization Arguments refactor type contributes to reliability because arguments such as maximum iterations and maximum retries improve the resilience of agent execution. The Compile with Checkpointer refactor contributes to reliability by providing a snapshot of agent state and execution to fall back to upon failure. Without the implementation of a checkpointer, the runtime state and execution history are all maintained solely within memory, so if the application crashes, a user will lose all of the data from their current thread. By saving this data to disk incrementally developers can provide a checkpoint to fall back to allowing the application to fail gracefully. LangChain's structured tool object allows developers to specify input/output schema when defining a tool. This can enhance the reliability of tools with complex inputs or outputs. The @tool decorator allows for similar assurance of inputs and outputs in less complex use cases. The tool decorator uses the tool function's docstring as a tool description providing agents with more context surrounding the tool's intended use-case. As a result, models are less likely to misuse or incorrectly select tools for a given user query. Additionally, the tool decorator uses the function's parameter and return type hints in the function signature to prevent type errors associated with agent tool use. The model fallbacks refactor is directed solely at reliability. Particularly when agentic applications are calling external APIs for model use, this refactor serves as an important mechanism making the application resilient in the event a certain model API or provider fails to respond to requests. The Specialized Model Routing refactor can optimize the tradeoff between cost and reliability by routing to state of the art models when queries are determined to be complex or crucial. This can ensure accurate model results while mitigating cost when possible. As a result, the Specialized Model Routing refactor impacts both cost and reliability.

n) Scalability Enhancing Refactorings. Scalability was addressed in 52 (30.6%) true refactorings making it the third most frequently addressed technical debt category. As depicted in Table 2 the distinct refactoring types affecting scalability include: ChatPromptTemplate, Alter State Schema to use add messages reducer, Reorganize Agent into Hierarchical Sub-Agents, Model Initialization Arguments, Compile with Checkpointer, and Minimize State Schema. The use of an add messages reducer in state enhances scalability by coordinating seamless, concurrent updates to state. This improves scalability by allowing multiple nodes running in parallel to write new data or metadata to the shared state without blocking one another. The Reorganize Agent into Hierarchical Sub-Agents refactor contributes to scalability by dividing a singular "God" agent into a supervisor(s) and single-responsibility sub-agents. This can help a system to scale with an increase in data or queries on a single thread because data context is more isolated depending on the specific task requested. This optimization is limited in the event that every request falls to the same sub-agent and the supervisor or delegator node may become a potential bottleneck depending on the configuration. The Minimize State Schema refactor improves the scalability of a langgraph application by reducing the amount of data maintained throughout graph execution. When state schema contains unnecessary fields the data that is recorded with each turn can quickly grow in multi-turn threads and make it difficult to scale an instance over many turns for a user. Implementing a checkpointer is essential to running a langgraph application at scale. LangGraph checkpointers are designed to save state into a thread as defined by a unique thread id, this in turn allows for different users or conversations to occur simultaneously without interfering with one another. Additionally, a checkpointer enables scaling across multiple workers such that any worker can pick up a request for a given thread by simply loading the most recent checkpoint and continuing to process a user request from that state. A checkpointer is also essential for human-in-the-loop workflows that pause and wait for further user input before proceeding with graph execution.

n) Understandability. Understandability is addressed by 21 commits (12.4%) in the true refactoring dataset. Among these, *Graph Restructure* is the most common refactoring type contributing to enhanced understandability, appearing in 8 of the 21 commits related to understandability. Restructuring a state graph to eliminate unnecessary conditional routing and minimize complexity improves understandability by making simpler and more comprehensible execution flows. In the event that a graph is large, complex, or contains many conditional edges it may take developers a great deal of effort to reconstruct the graph externally for maintainability or documentation purposes. Redesigning graphs to be more minimalistic can reduce the cognitive load associated with understanding the source code of a langgraph application. The other refactoring types contributing to understandability include: Consistent Node Naming Conventions, Consistent Tool Naming Conventions, and Modularization of Components. Among these Consistent Tool Naming Conventions is the least frequent refactoring type occurring only once in the true refactoring dataset. It is worth noting that most commits have consistent tool naming conventions prior to the changes. Proper node naming conventions can also enhance the understandability of a langgraph implementation by being self-descriptive and action-oriented such that developers reading the source code can immediately understand a node's purpose upon viewing it. The Modularization of Components refactor contributes meaningfully to understandability because it allows developers to easily locate and distinguish different components based on the file in which they are contained. A meaningfully organized file structure can make the purpose and classification of a component immediately apparent.

n) Technical Debt Co-Occurrences. Within true positive commits, however, reliability and maintainability were the most frequent occurring 68 and 64 times respectively. However, it is important to note that these two technical debts co-occur with one another 25 times in the true positive dataset. This means that 36.7% of reliability and 39% of maintainability occurrences address both of these technical debts simultaneously. This is because improved reliability and maintainability are often a by-product of refactorings whose explicit motivation is to address one of the two. It is also important to recognize that many of the identified refactorings address technical debt across multiple categories.

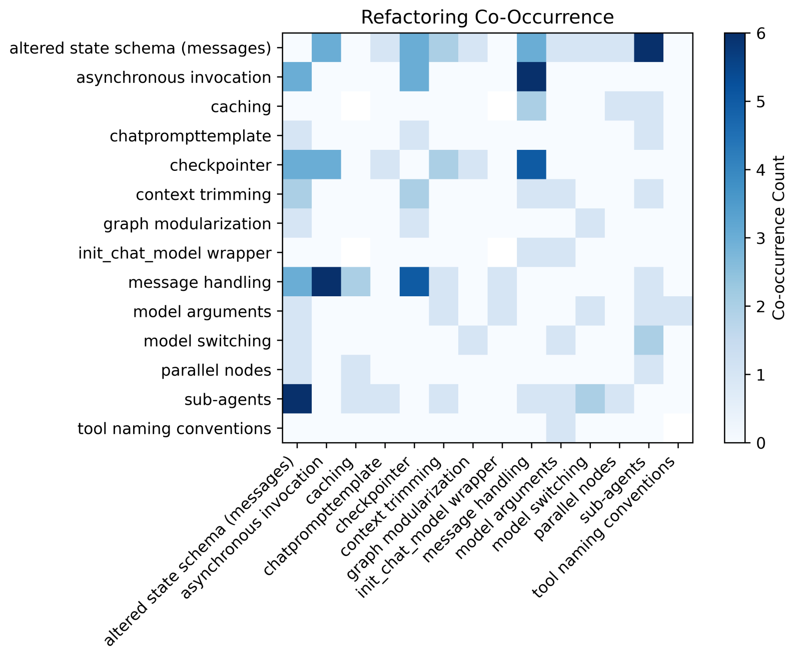

Fig. 1. Refactoring Co-Occurrence Heatmap

For instance the use of a prompt template improves both reliability and maintainability by improving language model responses and making prompt reuse seamless throughout application source code seamless for developers. Model initialization arguments are another versatile refactoring that can effectively impact the reliability, scalability, cost, performance, and compatibility of an application by setting explicit guardrails for language models and allowing for provider-specific keyword arguments.

n) Refactoring Type Co-Occurrences. Certain refactoring types frequently occur with others in the same commit. This happens, in part, because certain refactors mutually benefit from one another. The addition of the add messages reducer refactor and the hierarchical sub-agents refactor co-occur 6 times in the true positive dataset. This relationship exists because the add messages reducer function allows for concurrent state updates. When an agent is subdivided into multiple sub-agents the resulting graph will now benefit from concurrent updates in the event multiple agents are executing in parallel. Previously, the graph may have followed a simple sequential architecture without the need for complex state management; however, multiple agent nodes necessitate concurrent state management. A similar relationship exists between the Asynchronous Invocation and Message Handling refactoring types which occur in tandem 6 times as well. These refactoring types seem to have two distinct unrelated motivations and enhancements and yet developers are compelled to implement them within the same commit at a relatively high rate. The Message Handling and Checkpointer refactoring types have similar indicators of a relationship, occurring together 5 times in the dataset. This relationship is intuitive because as developers are implementing a checkpointer to improve memory persistence this refactor provides visibility to the quality of message management. That is to say that the contents of

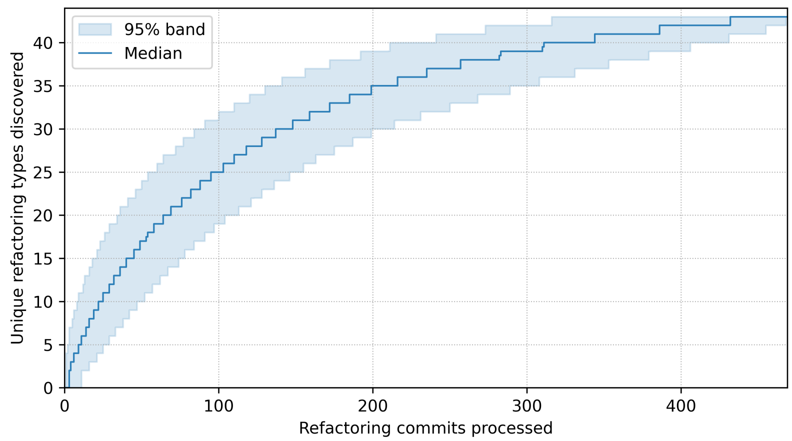

Fig. 2. Coverage of the Refactoring-Type Catalog as Data Scale Increases

memory becomes a concern when improving the memory implementation. Such changes have generally been observed in complete overhauls of memory management systems within the langchain portion of applications.

n) LangChain-Specific Refactorings. Table 3 depicts each extracted refactoring type along with the component affected and a short description of how the change is implemented. Each refactoring type affects one of eight distinct langchain components. The chain and graph component were grouped together because they each serve as the primary route for smaller interconnected components and vary only based on the scale or objective of the application. The messages component refers to message history across turns including system, human, and ai generated message objects. Messages refers specifically to conversational history or the management of message objects; this differs from other components such as state or memory. State is another distinct component that refers to the meta data passed between nodes in a langgraph state graph. The contents of state is entirely dependent upon how a developer chooses to implement it, but it is generally best practice to construct state as a typed dictionary. Model refers to the language model object which is invoked by other langchain components (e.g, nodes) typically with context such as previous messages and metadata. Rather than interacting with LLM provider APIs directly, developers generally use langchain's prebuilt model objects to leverage the abstraction they provide. The imports component refers to the exact langchain modules developers choose to import. There is some degree of overlap in the functionality offered by different langchain modules and as such developers can achieve the same functionality with different modules. Tools the most frequently impacted component across distinct refactoring types appearing in 5 of the 30 refactoring types (16.7%). Tools are langchain objects that provide models with functionality or allow a model to interact with other existing systems. Tools vary greatly depending upon the implementation but the same underlying refactorings occur throughout repositories. Individual nodes are a component impacted in only two refactorings and this is the only component that is framework specific, existing only in langgraph implementations.

n) Diversity of Refactoring Types Over Data Acquisition. The process of data acquisition and manual labeling of commits from various projects is intensive. One might argue that with an increase in our sample size, more unique refactoring types could be discovered. Figure 2 displays the justification of our sample size. This figure displays the median and 95% confidence interval of refactoring type discovery over the number of commits processed when shuffling and resampling the dataset 2000 times. This indicates that regardless of order the number of unique refactoring types discovered per new commit processed greatly decreases as the number of commits processed increases. Visually, this is represented by a significant decrease in slope as x increases. Thus, the evidence suggests that our team has extracted the majority of unique refactoring types present in open source langchain repositories.

Table 1. LangChain-Specific Technical Debts

Technical Debt	**Goal**	**Situation**	**Consequence**	**Intent Groups**
Cost	Minimize the monetary cost associated with LLM API calls	LLM calls are made too frequently or with many input tokens	User interaction is expensive due to over-reliance on API calls	Reduce input tokens; Avoid iterating uncontrollably; Prevent unnecessary API calls
Performance	Minimize the perceived latency	Graph or chain takes excessive time to respond/act when prompted	User experience is negatively impacted	Maximize cache reuse; Execute components asynchronously; Prevent unnecessary API calls; Enable parallelism
Maintainability	Reduce future change effort and change-ripple while preserving behavior	Evolution tasks (e.g., updating bases/packages, changing defaults, parameterizing build/run options) currently require repeated or fragile edits across artifacts	cascading edits, higher regression risk, longer lead time for change	Divide components into isolated files; Leverage wrappers to avoid provider-specific configurations; Use LangChain message-handling abstractions;
Reliability	Mitigate errors and undesired behavior	Errors, confusion and unexpected behavior	Errors and unexpected or undesirable behavior	Enhance type safety; Expand agent context; Increase agent determinism
Scalability	Seamlessly scale with an increase in users or data	The system cannot sustain an increase in users or data	Increase in users or data causes errors, latency, or expensive operations	Limit context/state in multi-turn threads; Divide graph into sub-agents or sub-graphs
Compatibility	Integrate seamlessly with new providers and dependencies	Updating dependencies, platform versions, or runtime environments causes failures due to hard-coded assumptions or environment-specific configurations.	Changes in dependencies or platforms causes errors or undesired behavior	
Understandability	Readable and comprehensible code for iterative development	Ambiguous naming conventions, monolithic code, or convoluted structure make it difficult for developers to interpret the code base.	Reading and understanding the project source code requires additional time and effort.	Consistent component naming conventions; Divide components into isolated files; Include type hints; Include descriptive doc-strings

Table 2. Mapping LangChain Refactoring Types to Technical Debt Categories.

Refactoring Type	Commit Count	Cost	Performance	Reliability	Scalability	Maintainability	Understandability	Compatibility
Asynchronous Invocation	10		×					
Adopt ChatPromptTemplate	6			×	×	×		
Alter State to Use Messages Reducer	24		×		×			
Decompose Agent into Sub-Agents	16				×			
Message Handling	40			×		×	×	
Initialize Model via Universal Interface	4					×		
Cache Recent Query Responses	8	×	×					
Add Model Arguments	10			×	×	×		×
Compile Graph with Checkpointer	24			×	×			
Migrate to Core APIs	4					×		
Minimize State Schema	3		×		×			
Migrate to StructuredTool	4			×				
Initialize Tools with Decorator	4			×			×	
Standardize Node Naming	2						×	
Standardize Tool Naming	2						×	
Context Trimming or Summarization	7	×	×					
Execute tool calls in parallel	2		×					
Execute nodes in parallel	5		×					
Model Fallbacks	2			×				
Add Model Routing Logic	5	×	×	×				
Modularization of Components	6					×	×	
Restructure Graph Topology	8		×				×	

4.2 Qualitative Analysis

The qualitative analysis of this study aims to analyze some valuable examples of commits that express the refactorings discussed in the previous section. While it is valuable to note that the refactorings presented are clear repeated patterns across open-source repositories, it is still vital that we evaluate what changed in the source code before and after a valid refactoring. This section provides brief static analysis for a variety of commits from the different categories mentioned previously. Here we provide more depth as to what changed before and after given refactorings as well as why developers claim it alleviates technical debt or enhances the code base.

Example 1 (Maintainability + Scalability + Performance - Use an annotated list with add messages reducer in graph state): Developers have a great deal of autonomy in designing the state schema for their state graph in a LangGraph application. The refactoring displayed in Listing 1 aims to follow the best practice for maintaining conversational memory in the graph state per official LangGraph documentation. By creating an annotated list of BaseMessage objects with the add messages reducer, developers take advantage of the abstraction provided by LangGraph objects while simultaneously allowing for concurrent updates to the list of BaseMessage objects. By altering the state schema to follow

Table 3. LangChain-Specific Refactorings

Refactoring Type	Component	Description
Asynchronous Invocation	Graph/Chain	Graphs or Chains/Sequences are invoked asynchronously using the ainvoke(), astream(), or abatch() method to prevent blocking
ChatPromptTemplate	Messages	Change prompt engineering to leverage ChatPromptTemplate rather than manually appending dictionary
Add Messages Reducer	State	Maintain conversational history with an annotated list of BaseMessage objects and corresponding add_messages reducer
Sub-Agents	Graph/Chain	Reconstruct graph into specialized sub-agents with supervisor/delegator node(s)
Message Handling	Messages	Alter message handling to leverage BaseMessage objects and their associated metadata
Initialize Model with Generic Interface	Model	Initialize model with generic init_chat_model rather than proprietary model wrapper
Cache Recent Query Responses	Messages	Cache recent queries to reduce latency & cost on repeat queries
Robust Model Initialization Arguments	Model	Initialize model with max iterations, max tokens, timeout, temperature, top-p, and model fallbacks as necessary
Compile Graph with Checkpointer	Graph/Chain	Initialize a memory saver & unique thread id then compile your stategraph with the checkpointer
Migrate off of community modules	Imports	Migrate off of community modules when identical functionality is avaliable in official modules
Minimize State Schema	State	Remove unused or unnecessary fields from state to reduce the amount of data transmitted between nodes
Structured Tool	Tools	Alter tool definitions to use StructuredTool for more robust input/output schema definitions
@tool decorator	Tools	Alter tool definitions to use the @tool decorator along with type hints and a descriptive docstring
Consistent Node Naming Conventions	Nodes	Alter node definitions to follow a clear & consistent naming convention in which the node name corresponds to the function name
Consistent Tool Naming Conventions	Tool	Alter tool definitions to follow clear & consistent naming conventions
Context Trimming & Summarization	Messages	When passing previous conversation turns to model trim down to the last n messages or summarize
Parallel Tool Calls	Tools	When a single query requires multiple tool calls, execute them in parallel
Parallel Nodes	Nodes	Reconstruct graph to allow for parallel execution of independent nodes
Specialized Model Routing	Model	Tailor model selection based on query analysis to ensure reliable results or minimize cost
Alter node returns to only updated fields	Node	Alter node functions to return only updated fields of state rather than the entire state
Use MCP for Tools	Tools	Use MCP as a tool service rather than LangChain tool objects
Migrate to Runnable Sequence	Chain	Migrate from chain functions to Runnable Sequence
Streaming	Graph/Chain	Stream graph/chain rather than invoking so tokens arrive as they are produced

this pattern developers aren't burdened by manually appending conversational history to a dictionary throughout the execution of the graph. Additionally, synchronous state updates can introduce some degree of reducible latency if

Listing 1. Commit ad59db5 from data-cleaning-agent: Use Annotated types for concurrent message updates

```
@@ -37,8 +38,8 @@ class DataCleaningState(TypedDict):
     pending_tasks: List[str]
     progress_percentage: float

-    # Agent communication
-    messages: List[BaseMessage]
+    # Agent communication - use add_messages for concurrent updates
+    messages: Annotated[List[BaseMessage], add_messages]
     agent_results: Dict[str, Dict]
     communication_log: List[Dict]
```

multiple tools, nodes, or agents are triggered simultaneously as state updates are required throughout the execution flow of the graph. Concurrent updates to the message state allow for a more seamless execution flow throughout the graph that minimizes blocking and allows for components to make concurrent progress without compromising state integrity.

Listing 2. Commit 84365f4 from hikizan-emacs: Integrated ‘langgraph.checkpoint.memory.MemorySaver‘ for persistent agent state across runs.

```
@@ -321,9 +418,18 @@ def __init__(self):
             google_api_key=GOOGLE_API_KEY,
         )

+        self.memory = MemorySaver()
+
         self.tools = [execute_elisp_code, read_file, write_to_file, list_files, grep, find_files]
         self.tools_by_name = {tool.name: tool for tool in self.tools}
-        self.graph = self._build_graph()
+
+        # Build graph with memory checkpointer
+        self.graph = self._build_graph().compile(checkpointer=self.memory)
```

Example n (Reliability + Scalability - Compile with checkpointer): The Listing 2 above exemplifies the compile with checkpointer refactoring. This refactoring can be performed with minimal changes to the code base. Developers can simply instantiate a LangGraph memory saver object and pass this memory saver as the checkpointer when compiling the graph into a runnable object. Previously, the graph was compiled with no functional checkpointer, so there exists no other version of state and runtime configuration to revert back to in the event of an error. For this reason, the checkpointer refactoring is classified as a contribution to reliability. Additionally, a checkpointer benefits the scalability of a langgraph application in multiple facets. An available persistent checkpointer allows for the data generated through graph execution to be offloaded to disk when necessary rather than relying solely on state, which exists in main memory, to capture execution and conversational history. This enhancement allows for a LangGraph application to reasonably scale with an increase in data, such as in a long multi-turn interaction.

While certain aspects of execution history such as prior messages can be maintained in the graph state, this state is maintained entirely in main memory. Thus, if a particular instance of the LangGraph application were to terminate, all of it’s history would be lost. This is generally not practical for the deployment of software at scale. This is why the LangGraph SDK offers a checkpointer which can be easily instantiated and compiled with a graph. The introduction of a checkpointer and thread id allows for memory persistence across threads. This means that for a given thread there are a series of checkpoints that save the graph state and runtime configuration of the agentic system at a given point in time. This in turn enhances the scalability of a LangGraph system as it allows for multi-tenant applications where each user is the owner of one or more threads. Additionally, by compiling each instance of the state graph with a unique checkpointer and thread id this allows for persistent memory in the event the application crashes or loses connection.

Listing 3. Commit c72917f from cm3070-lawtime: Switch to asynchronous graph invocation

```
diff --git a/server/services/task_service.py b/server/services/task_service.py
index dba2fb6..d1bac37 100644
--- a/server/services/task_service.py
+++ b/server/services/task_service.py
@@ -139,7 +139,7 @@ async def propose_tasks(
         try:
             # Invoke the LangGraph agent - it will handle all validation and initialization
             logger.info(f"Invoking LangGraph agent with source_type: {source_type}")
-            final_state = graph.invoke(initial_state)
+            final_state = await graph.ainvoke(initial_state)

             # Extract proposed tasks from final state
             proposed_tasks = final_state.get("proposed_tasks", [])
```

Example n (Performance - Use asynchronous version of invoke): Both LangChain and LangGraph offer asynchronous versions of the stream, batch, and invoke methods. The invoke method is by far the most common method for calling a compiled state graph or chain. Calling a graph or chain asynchronously can benefit the execution of an agentic system and prevent blocking. If chains or graphs are invoked synchronously this means that when a latent task such as making an external LLM call is performed, the rest of the program is blocked and cannot make progress on other routines. Contrarily, when a graph or chain is invoked asynchronously, this allows the program to execute concurrently and in turn reduce latency by making progress on tasks while waiting on latent tasks such as LLM calls. Asynchronous methods in the LangChain ecosystem still require that node and tool functions are defined asynchronously such that they can be called as a co-routine.

Listing 4. Commit f4877dc from boss-bot: Enhanced tool definitions with the '@tool' decorator, improving type safety and documentation

```
+    """Collection of media inspection tools using modern @tool decorator"""

     @staticmethod
+    @tool
     async def extract_video_metadata(video_path: str) -> Dict[str, Any]:
-        """Extract comprehensive video metadata"""
+        """Extract comprehensive video metadata using ffprobe or similar tools
+
+        Args:
+            video_path: Path to the video file to analyze
+
+        Returns:
+            Dictionary containing video metadata including duration, resolution, codecs, etc.
+        """
```

Example n (Reliability + Maintainability + Understandability - Use the @tool decorator with type hints): The LangChain SDK supports a tool decorator that allows for developers to seamlessly define tools while simultaneously providing informative context for a model when calling upon it. When a model calls a tool that is defined simply as a function in the codebase, oftentimes these tools require a variety of parameters to be passed of particular types. If a model passes the incorrect data type as a particular argument to a tool function this will lead to a type error and the tool call will not be able to execute. The tool decorator not only defines the function as a tool object, but also uses the function signature to provide the model with type hints to improve the reliability of tool calls. Type hints included in the function signature for both parameter and return types will mitigate type errors associated with tool calls. Additionally, the docstring is used as a tool description which is also available to the LLM. Informative docstrings can improve agent tool selection such that the most desireable tools are selected given the context and query.

Example n: Scalability + Maintainability + Reliability: Subdivide agent into specialized sub-agents. In large LangGraph applications, it can be preferable to subdivide a singular 'God agent' into single-responsibility sub-agents with a supervisor or routing node that acts as a delegator. Sub-agents can have a system prompt tailored to their granular use case as well as a tool subset that negates tools outside of their scope. Delegating to specialized sub-agents or sub-graphs can also allow for the parallel execution of independent tasks. In the event that the router or supervisor node elects multiple tasks for a given query, these co-routines can run concurrently and return to the supervisor without blocking one another.

Listing 5 portrays some key differences before and after the subdivision into modular sub-agents. As can be seen in the diff, the original version of the state graph routed to simple tools based on the presence of hard-coded keywords. This is a fragile practice with very little flexibility to respond to user interaction. There are many edge cases in which a user may adequately articulate the task which they wish to accomplish and this original logic may still fail to produce the desired results. Given the ability of modern language models to capture query intention, it is optimal to allow an agent to reason about the user query, develop a course of action, and delegate the necessary tasks. The additions shown in listing 5 show that rather than routing manually on the basis of hard-coded keywords, we can instantiate a deep agent with access to a list of sub-agents for the purpose of delegation. Sub-agents are equipped with their own unique subset of tools and task-specific system prompt. This reduction in agent scope can enhance the reliability of agent actions and reduce the likelihood of context-overload for a particular model instance. The implementation of a hierarchical sub-agent architecture such as that depicted in Listing 5 allows for seamless expansion of system functionality with minimal changes to the main workflow. Sub-agents are isolated and modular, thus, to add new agent functionality a new sub-agent can be easily created in isolation and simply added to the list of sub-agents available to the supervisor node. This enhanced ability to improve or expand workflow functionality improves the maintainability of the LangGraph implementation.

Example n: Reliability - Reduce temperature to increase model determinism. Listing 6 portrays an example of the temperature lowering refactoring. In this commit, the developer lowered the temperature value from 0.7 to 0.4 during model initialization. This is a common step taken to increase the determinism of agents. LLMs are not typically set to a default temperature of zero because this would indicate that the next generated token is always the most probable. Additionally, the developer chose to include a top-p value of 0.7 and decreased the max-tokens value by 50 percent. The top-p argument serves a similar purpose to temperature, in that it increases the determinism of model generation. Top-p (top probability) reduces the available sample size of a language model's probability distribution to only the tokens with the largest corresponding probabilities whose cumulative sum is equal to the top probability value. This

Listing 5. Commit cb3467c from hacs-ai: Complete replacement of basic graph with deepagents framework; 5 specialized admin sub-agents for different domains

```diff
diff --git a/examples/hacs_developer_agent/deep_agent.py b/examples/hacs_developer_agent/deep_agent.py
new file mode 100644
index 0000000..08a2217
--- /dev/null
+++ b/examples/hacs_developer_agent/deep_agent.py
@@ -0,0 +1,345 @@
+def create_hacs_deep_admin_agent(
+    instructions: str,
+    model: Optional[Union[str, LanguageModelLike]] = None,
+    subagents: List[SubAgent] = None,
+    additional_tools: List[Union[BaseTool, Dict[str, Any]]] = None,
+    database_url: Optional[str] = None,
+):
diff --git a/examples/hacs_developer_agent/graph.py b/examples/hacs_developer_agent/graph.py
index 2e84a99..ce15384 100644
--- a/examples/hacs_developer_agent/graph.py
+++ b/examples/hacs_developer_agent/graph.py
@@ -1,212 +1,87 @@
-async def admin_operation_router(state: State) -> Dict[str, Any]:
-    """Route to appropriate admin operation based on messages."""
-    global _current_operation, _operation_in_progress
-
-    # Get the last message to determine operation type
-    messages = state.get("messages", [])
-    if not messages:
-        return {"last_error": "No operation specified"}
-
-    last_message = messages[-1].content.lower()
-
-    # Determine operation type from user message
-    if any(word in last_message for word in ["migration", "migrate", "setup", "initialize"]):
-        operation_type = "database_migration"
-    elif any(word in last_message for word in ["status", "check", "migration status"]):
-        operation_type = "migration_status"
-    elif any(word in last_message for word in ["schema", "describe", "tables", "structure"]):
-        operation_type = "schema_inspection"
-    elif any(word in last_message for word in ["create", "record", "resource"]):
-        operation_type = "create_resource"
-    elif any(word in last_message for word in ["discover", "find", "available"]):
-        operation_type = "discover_resources"
-    else:
-        operation_type = "general_help"
-
-    # Set private tracking variables
-    _current_operation = operation_type
-    _operation_in_progress = True
-
-    return {}  # No state changes, just private tracking
```

means that the probability distribution is truncated such that as soon as the cumulative probability of the most probable tokens equals or exceeds the top-p value, all other tokens are dropped. In short, lowering temperature decreases the degree to which randomness is introduced in token sampling, while lowering top probability decreases the sample size, making it impossible to sample highly improbable tokens. If agents are interacting frequently with tools, the mitigation of unusual tokens will greatly reduce the likelihood of type errors and tool misuse. Thus, well configured temperature and top probability arguments can improve the reliability of LangChain and LangGraph applications.

Listing 6. Commit 0ad71e1 from ai-resume-agent: Maintain temperature=0.1 for consistent responses; Configure ChatGroq with top_p parameter to reduce creativity

```
diff --git a/app/core/config.py b/app/core/config.py
index 3f4f21c..b5a276c 100644
--- a/app/core/config.py
+++ b/app/core/config.py
@@ -2,68 +2,90 @@
-    GROQ_API_KEY: str
-    GROQ_MODEL: str = "llama-3.3-70b-versatile"  # Modelo actualizado
-    GROQ_TEMPERATURE: float = 0.7
-    GROQ_MAX_TOKENS: int = 1024
-    GROQ\_TIMEOUT: int = 30  # Timeout en segundos (protecci n anti-DoS)
+    # Google Gemini API (LLM alternativo)
+    GEMINI_API_KEY: str = ""
+    GEMINI_MODEL: str = "gemini-2.5-flash"  # Modelo m s r pido y menos restrictivo
+    GEMINI_TEMPERATURE: float = 0.4  # M s confianza para sintetizar respuestas STAR
+    GEMINI_TOP_P: float = 0.7  # Ventana m s amplia para construcci n de frases
+    GEMINI_MAX_TOKENS: int = 512  # Espacio suficiente para respuestas detalladas
```

In the context of a chat bot it may be ideal to introduce a degree of randomness in order to produce more natural language. However, in certain agentic systems, particularly those with tools or specific input and output schemas, developers seek to create a more deterministic system in which the most likely output is typically the desired one. Setting the temperature of a model to zero maximizes the determinism of a model and leads to behavior in which the LLM will always produce the next token with the highest associated probability. With a temperature set to zero, language models will pass arguments to tools with the highest associated probability given the context of the query as well as tool descriptions and type hints. This can enhance the reliability of tool results and potentially mitigate type errors.

Example n: Context trimming for multi-turn threads. When maintaining conversational history in a LangChain or LangGraph application it is common for context to be fed back into the model along with the current query to provide the model with a broader context including previous queries and responses. The act of feeding previous turns back into the model increases the number of tokens being processed by the model and inherently the cost per query. Particularly in the event a conversation between a user and agent takes many turns this will cause the cost of each subsequent query to grow uncontrollably. In extreme cases this may lead to a user exceeding the context window of the LLM. This commit exemplifies a common strategy among developers to trim conversational context to the last n messages. This allows developers to manage the tradeoff between context and cost. It may be that conversational history between the user and agent can enhance the accuracy of future responses at the expense of increased token consumption.

Example n: Prompt Engineering with ChatPromptTemplate. The ChatPromptTemplate object allows developers to organize messages in a multi-turn conversation as a dictionary of strings associated with the correct role. System prompts are correctly identified as such. User generated queries are assigned the role "user" and model generated responses are assigned the role key "assistant". This is advantageous during a multi-turn conversational workflow because it effectively differentiates between each turn and assigns that string to the correct role thus improving model comprehension of context. The prompt template also enables consistent formatting of model responses. ChatPromptTemplate adds support for placeholders allowing for the insertion of context specific data during runtime. The use of a prompt template enables developers to capture context over the course of a multi-turn interaction in order to enhance model awareness.

Listing 7. Commit 84365f4 from hikizan-emacs: Implemented message trimming (‘trim_messages_window‘) and memory optimization to manage context window size and prevent overflow.

```
diff --git a/python/emacsagent.py b/python/emacsagent.py
index 4b9308f..11ba7be 100644
--- a/python/emacsagent.py
+++ b/python/emacsagent.py
@@ -23,6 +23,8 @@
 MODEL_NAME = "gemini-2.5-flash"
 LLM_TEMPERATURE = 0
 TIMEOUT = 60
+MEMORY_WINDOW_SIZE = 10  # Keep last 10 messages in active memory
+MAX_EXECUTION_LOG_SIZE = 50  # Keep last 50 execution steps
@@ -116,6 +118,100 @@
+def trim_messages_window(messages: Sequence[BaseMessage], window_size: int = MEMORY_WINDOW_SIZE) ->
     Sequence[BaseMessage]:
+    """
+    Trim messages to keep only the most recent ones within the window size.
+    Always preserve the system message if it exists.
+    """
+    if len(messages) <= window_size:
+        return messages
+
+    # Check if first message is system message
+    if messages and isinstance(messages[0], SystemMessage):
+        # Keep system message + last (window_size - 1) messages
+        return [messages[0]] + list(messages[-(window_size-1):])
+    else:
+        # Just keep last window_size messages
+        return list(messages[-window_size:])
```

Example n: Initializing models with generic model wrapper for enhanced maintainability. Listing 9 depicts a self-admitted refactoring in which the author simplifies the process of model initialization by using LangChain's generic init_chat_model object. This refactoring allows them to remove three helper functions associated with each of the proprietary providers available in their application. The improved implementation leverages a generic interface while still allowing the flexibility of passing provider-specific arguments via the kwargs field. The removal of hard-coded logic and provider specific functions not only streamlines the function but also expands the number of configurable models to all of those supported by the init_chat_model interface. If a developer wished to add another model option to the application in the previous implementation, they would have to build yet another function and all of the necessary implementation for those unique provider fields. In the new implementation, most mainstream models are available by default.

Example n: Caching model responses for repeat queries. In Listing 10 the author claims to have produced a significant latency reduction by caching cover-letter query responses with a time-to-live of eight hours. In their commit message they explicitly claim to have reduced latency on repeated cover-letter queries from 8 seconds to 3 milliseconds. Although it is difficult to validate this claim without completely cloning the repository and environment, we can logically conclude that caching responses would reduce latency and cost by avoiding an unnecessary external call to a language model. However, this must be done at the expense of increased resources to maintain cached responses for 8 hours at a time. Nonetheless, systems that are prone to receive repeat queries from users and that prioritize quick response time over optimized memory usage will benefit from such an implementation.

Listing 8. Commit 447a09c from file-classifier: refactor: migrate prompt management to LangChain ChatPromptTemplate

```
diff --git a/src/ai_file_classifier/ai_client.py b/src/ai_file_classifier/ai_client.py
index c757756..5a79f45 100644
--- a/src/ai_file_classifier/ai_client.py
+++ b/src/ai_file_classifier/ai_client.py
@@ -133,15 +134,15 @@ def _initialize_llm(
-        system_prompt: str,
-        user_prompt: str,
+        prompt_template: ChatPromptTemplate,
+        prompt_values: dict,
diff --git a/src/ai_file_classifier/prompt_manager.py b/src/ai_file_classifier/prompt_manager.py
new file mode 100644
index 0000000..72a5851
--- /dev/null
+++ b/src/ai_file_classifier/prompt_manager.py
@@ -0,0 +1,73 @@
+def load_file_analysis_prompt() -> ChatPromptTemplate:
+    """
+    Load the file analysis prompt template from text files.
+
+    Returns:
+        ChatPromptTemplate: A LangChain prompt template with system and human messages.
+
+    Raises:
+        FileNotFoundError: If prompt files are missing.
+        IOError: If prompt files cannot be read.
+    """
+    try:
.......
+        # Create ChatPromptTemplate with proper message roles
+        return ChatPromptTemplate.from_messages([
+            ("system", system_prompt),
+            ("human", user_prompt),
+        ])
```

Example n: Return only state updates from nodes. Per LangGraph documentation nodes are intended to return only the updated fields of state as a dictionary. These updates are then merged into the existing state by the LangGraph engine. Depicted in Listing 11 is a refactor in which the author corrected their node function to follow this pattern. Direct mutations to state can appear to work in some simple graphs that aren't concerned with previous iterations, however, there are some major risks that can become apparent in complex applications. Firstly, when directly mutating fields in the state you override any previous values essentially deleting any data from previous nodes or iterations. This practice may actually function if the program is only ever concerned with the state from the previous step, but it does not fully encompass the ability of state to maintain data throughout graph execution. Additionally, if a graph contains branching and parallelism there is a major risk of overriding state updates from parallel processes. If some state updates are completely overridden and lost, this can hinder graph execution that relies on aspects of state down the line because now there is missing data.

Example n: Limit API calls to reduce cost and latency. As we defined earlier cost as technical debt in the scope of this paper refers to the monetary cost associated with large language model queries. We can reduce this cost by either reducing the number of tokens present within queries or by reducing the number of queries made. Depicted in Listing 12 is a refactor that elects to limit the number of API calls made to an external provider (OpenAI in this case) by implementing a conditional edge with simple routing logic. The _should_continue_after_tools function acts as the logic

Listing 9. Commit 8c4678d from diversifier: Refactor llm_factory.py to use LangChain's init_chat_model for unified provider initialization

```
diff --git a/src/orchestration/llm_factory.py b/src/orchestration/llm_factory.py
index aa13ef8..c84ce2d 100644
--- a/src/orchestration/llm_factory.py
+++ b/src/orchestration/llm_factory.py
@@ -27,153 +28,46 @@ def create_llm_from_config(config: Optional[LLMConfig] = None) -> Any:
-    # Base parameters common to most providers
-    params = {
-        "model_name": config.model_name,
-        "temperature": config.temperature,
-        "max_tokens": config.max_tokens,
-        "timeout": config.timeout,
-        **config.additional_params,
-    }
-    # Add base_url if specified
-    if config.base_url:
-        params["base_url"] = config.base_url
-    logger.info(
-        f"Creating LLM instance: provider={provider}, model={config.model_name}"
-    )
-    if provider == "anthropic":
-        return _create_anthropic_llm(config, params)
-    elif provider == "openai":
-        return _create_openai_llm(config, params)
-    elif provider == "google":
-        return _create_google_llm(config, params)
-    else:
-        raise ValueError(f"Unsupported LLM provider: {provider}")
.......
+        # Use init_chat_model with our configuration
+        from typing import Any
+
+        kwargs: dict[str, Any] = {
+            "temperature": config.temperature,
+            "max_tokens": config.max_tokens,
+        }
+        kwargs.update(config.additional_params)
+
+        return init_chat_model(model=model_id, **kwargs)
```

behind the conditional edge added to the workflow from tools that routes either back to the agent node for further reasoning or to the synthesize node to complete the workflow. In Listing 12 we observe simple if-else logic that routes directly to synthesize if the workflow has already successfully executed multiple successful tool calls or iterations. If neither of these conditions are met then the workflow does return to the agent node and proceeds to call the model for further reasoning. This function prevents uncontrollably iterating between tools and reasoning which may result in uncontrolled spending to the API and an increase in latency perceived by the user.

Example n: Initialization Arguments. This particular repository leverages a langchain prebuilt agent for even further abstraction. As such, the agent executor acts as an object similar to a model when bound with tools. The agent executor object accepts similar arguments to chat model wrappers such as maximum iterations. In this refactoring the developer elects to set maximum iterations to 10 where it was not previously. This makes it impossible for the agent to iterate uncontrollably between tools and the model, repeatedly reasoning about previous tool calls increasing cost and latency without guardrails. Setting the maximum iterations parameter to an integer depending on the use case of the

Listing 10. Commit 8108951 from aruizca-resume: Cover letter caching reduces 8s operations to 3ms (cache hits)

```
diff --git a/packages/core/src/main/cover-letter/infrastructure/langchain/CoverLetterPromptRunner.ts b
     /packages/core/src/main/cover-letter/infrastructure/langchain/CoverLetterPromptRunner.ts
index 586671d..18ca0db 100644
--- a/packages/core/src/main/cover-letter/infrastructure/langchain/CoverLetterPromptRunner.ts
+++ b/packages/core/src/main/cover-letter/infrastructure/langchain/CoverLetterPromptRunner.ts
+      cacheConfig: {
+        ttl: 8 * 60 * 60 * 1000 // 8 hours
+      },
diff --git a/packages/core/src/main/shared/infrastructure/langchain/LangchainPromptRunner.ts b/
     packages/core/src/main/shared/infrastructure/langchain/LangchainPromptRunner.ts
new file mode 100644
index 0000000..74d22a5
--- /dev/null
+++ b/packages/core/src/main/shared/infrastructure/langchain/LangchainPromptRunner.ts
@@ -0,0 +1,153 @@
+      // Check cache first
+      const cachedResponse = await this.cache.get(input, promptTemplateString, forceRefresh);
+      if (cachedResponse && !forceRefresh) {
+        return this.config.outputTransformer(cachedResponse);
+      }
+      // Execute the LLM operation with performance monitoring
+      const result = await performanceMonitor.trackOperation(
+        operationName,
+        async () => {
+          // Create and execute chain
+          const chain = await this.createChain();
+          return await chain.invoke(promptVariables);
+        },
+        { logToConsole: true }
+      );
+      // Cache the response
+      await this.cache.set(input, promptTemplateString, result);
```

Listing 11. Commit c373419 from JAA: Updated Node Return Values - All nodes now return only state updates (dictionaries) instead of mutating and returning full state objects

```
diff --git a/llm/llm.py b/llm/llm.py
index 4a6fea4..f2bf4d9 100644
--- a/llm/llm.py
+++ b/llm/llm.py
@@ -195,12 +214,14 @@ def _router_node(self, state: AgentState) -> AgentState:
         else:
             func_name = RouterFunction.HUMAN_TEXT
+            query     = ""
-        state["router_decision"] = func_name
-        state["router_query"]    = query
-        state["query_keywords"]  = keywords
-
-        return state
+        # Return only updates - LangGraph will merge into state
+        return {
+            "router_decision": func_name,
+            "router_query":    query,
+            "query_keywords":  keywords
+        }
```

application is a near effortless action that prevents behaviors that can negatively impact the cost and latency of an agentic application.

Listing 12. Commit 0760805 from cybersheild: Reduce OpenAI API calls by 75% (from 4-8+ to 1-2 calls per analysis); Implement smart synthesis routing after sufficient tool results

```
diff --git a/workflows/react_workflow.py b/workflows/react_workflow.py
index d22f0e1..915e359 100644
--- a/workflows/react_workflow.py
+++ b/workflows/react_workflow.py
  @@ -73,17 +83,24 @@
  +        # Add edges - optimized to reduce OpenAI API calls
         workflow.set_entry_point("agent")
         workflow.add_conditional_edges(
             "agent",
             self._should_continue,
             {
-                "continue": "tools",
-                "end": "synthesize"
+                "tools": "tools",      # Go to tools if we have tool calls
+                "synthesize": "synthesize"  # Go directly to synthesis if final answer
+            }
+        )
+        workflow.add_conditional_edges(
+            "tools",
+            self._should_continue_after_tools,
+            {
+                "agent": "agent",      # Only go back to agent if more reasoning needed
+                "synthesize": "synthesize"  # Otherwise synthesize directly
             }
         )
-        workflow.add_edge("tools", "agent")
@@ -370,21 +387,49 @@
+    def _should_continue_after_tools(self, state: CyberShieldState) -> str:
+        """Decide whether to continue reasoning or synthesize after tool execution"""
+        iteration = state.get("iteration_count", 0)
+        scratchpad = state.get("agent_scratchpad", "")
+        # Count successful tool executions
+        successful_observations = scratchpad.count("Observation:") - scratchpad.count('"error"')
+        # If we have multiple successful tool results, go straight to synthesis
+        if successful_observations >= 2 or iteration >= 1:
+            logger.info("Moving to synthesis after tools",
+                        successful_observations=successful_observations,
+                        iteration=iteration,
+                        reason="sufficient_data")
+            return "synthesize"
+        elif iteration > 5:  # Hard limit to prevent loops
+            logger.info("Forcing synthesis due to iteration limit", iteration=iteration)
+            return "synthesize"
         else:
-            return "continue"
+            logger.info("Continuing to agent for more reasoning", iteration=iteration)
+            return "agent"
```

Example n: Tool Correction with descriptive naming & docstring. As described in Table 3, the tool decorator from LangChain is designed to use the Docstring of the tool function as a description for the model. This allows the model to reason about the tool's functionality and determine if it's use-cases align with the query. As a result, developers are encouraged to make a functionally descriptive docstring with the elements "Args" and "Returns" included. In the event that the model does elect to execute this tool, it will now be informed of the arguments that it needs to pass and what is expected to be returned. This greatly enhances the reliability of model actions because it reduces the likelihood

Listing 13. Commit e50b85e from ai-cv-agent: Refactor LangChain CV Agent

```
diff --git a/agent/langchain_cv_agent.py b/agent/langchain_cv_agent.py
index 20dc2b4..cb0eddd 100644
--- a/agent/langchain_cv_agent.py
+++ b/agent/langchain_cv_agent.py
@@ -235,32 +201,51 @@
-        return AgentExecutor(agent=agent, tools=self.tools, verbose=True)
+        return AgentExecutor(
+            agent=agent,
+            tools=self.tools,
+            verbose=True,
+            max_iterations=10,
+            handle_parsing_errors=True,
+            return_intermediate_steps=True  # Enable to see artifact passing
+        )
```

Listing 14. Commit e50b85e from ai-cv-agent: Refactor LangChain CV Agent

```
diff --git a/agent/langchain_cv_agent.py b/agent/langchain_cv_agent.py
index 20dc2b4..cb0eddd 100644
--- a/agent/langchain_cv_agent.py
+++ b/agent/langchain_cv_agent.py
@@ -51,179 +57,139 @@
-        @tool
-        def analyze_job_posting(job_url: str) -> str:
-            """Analyze a job posting and extract key requirements, skills, and keywords"""
+        @tool(response_format="content_and_artifact")
+        def analyze_job_with_llm(job_description: str) -> tuple[str, dict]:
+            """
+            Analyze job posting to extract key requirements and keywords.
+            Args:
+                job_description: The job description text
+            Returns:
+                Tuple of (analysis summary, structured analysis dict as artifact)
+            """
```

of a model passing an inaccurate argument or misinterpreting the values returned by a tool. Shown in Listing 14 is a refactoring in which the developer makes such a correction to their tool. In the original implementation there are type hints in the function signature which is ideal to enhance type safety, but the docstring lacks nuance. Among some other alterations to the tool, the commit author adds associative "Args" and "Returns" fields to enhance model awareness. Additionally, the author altered the name of the tool to be more descriptive which also enhances understandability for developers working on the source code.

Example n: Setting maximum tokens .

Example n: Implement message trimmer for context limits. Listing 16 shows a refactoring in which the developer utilizes LangChain's prebuilt trim_messages and instantiates the trimmer with maximum tokens set to 10 and the strategy set to last, such that only the last 10 tokens are preserved in conversational memory. This allows the developer to effectively minimize context in multi-turn interactions and limit cost as a result. It is important to express the trade-off associated with cost and context because increased context can improve the accuracy or reliability of model results at the expense of increased cost. The trim_messages feature offered by LangChain allows developers the opportunity to effectively manage this tradeoff depending on the circumstances of their application.

Listing 15. Commit b44b80a from medabot: Added maxTokens: 1000 to limit response length

```
diff --git a/app/core/leafletProcessor.ts b/app/core/leafletProcessor.ts
index c42a33e..ec51887 100644
--- a/app/core/leafletProcessor.ts
+++ b/app/core/leafletProcessor.ts
@@ -50,61 +78,94 @@
-     // Create chat model
+     // Use gpt-4o-mini - better quality, faster, and cheaper than gpt-4.1-nano
+     // gpt-4o-mini has 128k context window and better instruction following
      const llm = new ChatOpenAI({
-       modelName: "gpt-4.1-nano",
-       temperature: 0,
+       modelName: "gpt-4o-mini", // Updated from gpt-4.1-nano
+       temperature: 0.1, // Slightly increased for more natural language, but still mostly
     deterministic
        openAIApiKey: process.env.OPENAI_API_KEY,
+       maxTokens: 1000, // Limit response length
```

Listing 16. Commit e7cd19c from Agente-de-IA-usando-Next-y-Langchain: Introduce a trimmer to manage conversation history by trimming messages based on token count

```
diff --git a/lib/langgraph.ts b/lib/langgraph.ts
index 74df180..e076539 100644
--- a/lib/langgraph.ts
+++ b/lib/langgraph.ts
@@ -1,12 +1,20 @@
+// Create a trimmer to trim the messages
+const trimmer = trimMessages({
+  maxTokens: 10,
+  strategy: "last",
+  tokenCounter: (msgs) => msgs.length,
+  includeSystem: true,
+  allowPartial: false,
+  startOn: "human",
+});
+       // Trim the messages to manage conversation history
+       const trimmedMessages = await trimmer.invoke(state.messages);
+       // Format the prompt with the current messages
+       const prompt = await promptTemplate.invoke({ messages: trimmedMessages });
+       const response = await model.invoke(prompt);
+       return { messages: [response] };
```

Example n: Model routing based on perceived query difficulty. Our study has observed repeatedly that repositories in the open source community will often route to different large language models depending upon the user query. This can be done with one of two motivations. Firstly, some applications may route to specific models depending upon the task implicitly stated in a query to take advantage of the strengths of particular models. Secondly, a developer may intend to extract the difficulty of the query and route to a specific model based on that perceived difficulty; this is the scenario that is depicted in Listing 17. In this particular project the commit author has attempted to extract the difficulty of the query based on user input and route to a smaller, less intensive model if the agentic strength is perceived to be low or very low. In this particular implementation both models (gpt-4o and gpt-4o-mini) are provided by OpenAI, but the mini model allows for both a cost and latency reduction when queries are simple and extensive reasoning is not required. This is an optimization that allows for improved cost and performance as technical debt while also allowing the user the opportunity to use a more robust model for reliable results with more complex tasks.

Listing 17. Commit 231d4f2 from bt-servant-engine: honor the effective strength by switching to ‘gpt-4o-mini‘ when the level is low, and introduce the new ‘set-agentic-strength‘ intent that persists user requests

```
diff --git a/brain.py b/brain.py
index 2f6cd08..9f3868a 100644
--- a/brain.py
+++ b/brain.py
@@ -719,6 +735,68 @@
+def _model_for_agentic_strength(
+    agentic_strength: str,
+    *,
+    allow_low: bool,
+    allow_very_low: bool,
+) -> str:
+    """Return GPT model name based on strength and allowed downgrades."""
+    allowed: set[str] = set()
+    if allow_low:
+        allowed.add("low")
+    if allow_very_low:
+        allowed.add("very_low")
+    return "gpt-4o-mini" if agentic_strength in allowed else "gpt-4o"
@@ -1271,7 +1674,7 @@
     completion = open_ai_client.chat.completions.create(
-        model="gpt-4o",
+        model=model_name,
@@ -1356,7 +1764,8 @@
-    query_language = detect_language(query)
+    agentic_strength = _resolve_agentic_strength(s)
+    query_language = detect_language(query, agentic_strength=agentic_strength)
```

Listing 18. Commit 818d728 from agent_inbox: Handle both string and list content (multimodal messages); Type-safe content extraction for AIMessage and ToolMessage

```
diff --git a/src/graph.py b/src/graph.py
index a4eb0ce..aad0382 100644
--- a/src/graph.py
+++ b/src/graph.py
@@ -676,15 +676,21 @@
-        if isinstance(last_msg, AIMessage) and (
-            "recap" in last_msg.content.lower() or "summary" in last_msg.content.lower()
-        ):
-            return {"messages": messages}
+        # Handle both string and list content (multimodal messages with tool calls)
+        if isinstance(last_msg, AIMessage):
+            content = last_msg.content
+            if isinstance(content, str) and (
+                "recap" in content.lower() or "summary" in content.lower()
+            ):
+                return {"messages": messages}
         source_text = ""
         for m in reversed(messages):
             if isinstance(m, AIMessage) or isinstance(m, ToolMessage):
-                source_text = (m.content or "").strip()
+                # Type-safe content extraction for multimodal messages
+                content = m.content or ""
+                source_text = content if isinstance(content, str) else str(content)
+                source_text = source_text.strip()
```

Example n: Improved message handling. The LangChain handles messages as either HumanMessage, AIMessage, or ToolMessage objects, all of which extend the BaseMessage class. Each message type contains not only the contents of a query or response but also the necessary metadata associated with their execution. Corrections made to message handling is one of the most frequently observed patterns in LangChain refactorings. The refactor depicted in Listing 18 alters the post-model hook to handle messages more resiliently regardless of the message type. The refactored method allows for graceful handling of tool calls in the event that they are returned while still maintaining the ability to extract the contents of an AIMessage object. The improved implementation avoids type-errors in the event that tool calls or other metadata are contained within the message object.

5 Citations and Bibliographies

The use of BibTEX for the preparation and formatting of one's references is strongly recommended. Authors' names should be complete — use full first names ("Donald E. Knuth") not initials ("D. E. Knuth") — and the salient identifying features of a reference should be included: title, year, volume, number, pages, article DOI, etc.

The bibliography is included in your source document with these two commands, placed just before the `\end{document}` command:

```
\bibliographystyle{ACM-Reference-Format}
\bibliography{bibfile}
```

where "`bibfile`" is the name, without the "`.bib`" suffix, of the BibTEX file.

Citations and references are numbered by default. A small number of ACM publications have citations and references formatted in the "author year" style; for these exceptions, please include this command in the **preamble** (before the command "`\begin{document}`") of your LaTEX source:

```
\citestyle{acmauthoryear}
```

Some examples. A paginated journal article [2], an enumerated journal article [11], a reference to an entire issue [10], a monograph (whole book) [24], a monograph/whole book in a series (see 2a in spec. document) [18], a divisible-book such as an anthology or compilation [13] followed by the same example, however we only output the series if the volume number is given [14] (so Editor00a's series should NOT be present since it has no vol. no.), a chapter in a divisible book [36], a chapter in a divisible book in a series [12], a multi-volume work as book [23], a couple of articles in a proceedings (of a conference, symposium, workshop for example) (paginated proceedings article) [3, 16], a proceedings article with all possible elements [35], an example of an enumerated proceedings article [15], an informally published work [17], a couple of preprints [6, 8], a doctoral dissertation [9], a master's thesis: [4], an online document / world wide web resource [1, 28, 37], a video game (Case 1) [27] and (Case 2) [26] and [25] and (Case 3) a patent [34], work accepted for publication [31], 'YYYYb'-test for prolific author [32] and [33]. Other cites might contain 'duplicate' DOI and URLs (some SIAM articles) [22]. Boris / Barbara Beeton: multi-volume works as books [20] and [19]. A presentation [30]. An article under review [7]. A couple of citations with DOIs: [21, 22]. Online citations: [37–39]. Artifacts: [29] and [5].

6 Acknowledgments

Identification of funding sources and other support, and thanks to individuals and groups that assisted in the research and the preparation of the work should be included in an acknowledgment section, which is placed just before the reference section in your document.

This section has a special environment:

```
\begin{acks}
...
```

```
\end{acks}
```

so that the information contained therein can be more easily collected during the article metadata extraction phase, and to ensure consistency in the spelling of the section heading.

Authors should not prepare this section as a numbered or unnumbered `\section`; please use the "`acks`" environment.

7 Appendices

If your work needs an appendix, add it before the "`\end{document}`" command at the conclusion of your source document.

Start the appendix with the "`appendix`" command:

```
\appendix
```

and note that in the appendix, sections are lettered, not numbered. This document has two appendices, demonstrating the section and subsection identification method.

8 SIGCHI Extended Abstracts

The "`sigchi-a`" template style (available only in LaTeX and not in Word) produces a landscape-orientation formatted article, with a wide left margin. Three environments are available for use with the "`sigchi-a`" template style, and produce formatted output in the margin:

`sidebar`: Place formatted text in the margin.
`marginfigure`: Place a figure in the margin.
`margintable`: Place a table in the margin.

Acknowledgments

To Robert, for the bagels and explaining CMYK and color spaces.

References

[1] Rafal Ablamowicz and Bertfried Fauser. 2007. *CLIFFORD: a Maple 11 Package for Clifford Algebra Computations, version 11.* Retrieved February 28, 2008 from http://math.tntech.edu/rafal/cliff11/index.html

[2] Patricia S. Abril and Robert Plant. 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36–44. doi:10.1145/1188913.1188915

[3] Sten Andler. 1979. Predicate Path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226–236. doi:10.1145/567752.567774

[4] David A. Anisi. 2003. *Optimal Motion Control of a Ground Vehicle.* Master's thesis. Royal Institute of Technology (KTH), Stockholm, Sweden.

[5] Sam Anzaroot and Andrew McCallum. 2013. *UMass Citation Field Extraction Dataset.* Retrieved May 27, 2019 from http://www.iesl.cs.umass.edu/data/data-umasscitationfield

[6] Sam Anzaroot, Alexandre Passos, David Belanger, and Andrew McCallum. 2014. *Learning Soft Linear Constraints with Application to Citation Field Extraction.* arXiv:1403.1349 doi:10.48550/arXiv.1403.1349

[7] R. Baggett, M. Simecek, C. Chambellan, K. Tsui, and M. Fraune. 2025. Fluidity in the Phased Framework of Technology Acceptance: Case Study to Gain a Holistic Understanding of (Older Adult) Participant Advancement Through Acceptance Phases with Mobile Telepresence Robots. *Robotics Aut. Systems*. Manuscript submitted for review.

[8] Lutz Bornmann, K. Brad Wray, and Robin Haunschild. 2019. *Citation concept analysis (CCA)—A new form of citation analysis revealing the usefulness of concepts for other researchers illustrated by two exemplary case studies including classic books by Thomas S. Kuhn and Karl R. Popper.* arXiv:1905.12410 [cs.DL]

[9] Kenneth L. Clarkson. 1985. *Algorithms for Closest-Point Problems (Computational Geometry).* Ph. D. Dissertation. Stanford University, Palo Alto, CA. UMI Order Number: AAT 8506171.

[10] Jacques Cohen (Ed.). 1996. Special issue: Digital Libraries. *Commun. ACM* 39, 11 (Nov. 1996).

[11] Sarah Cohen, Werner Nutt, and Yehoshua Sagic. 2007. Deciding equivalances among conjunctive aggregate queries. *J. ACM* 54, 2, Article 5 (April 2007), 50 pages. doi:10.1145/1219092.1219093

[12] Bruce P. Douglass, David Harel, and Mark B. Trakhtenbrot. 1998. Statecarts in use: structured analysis and object-orientation. In *Lectures on Embedded Systems*, Grzegorz Rozenberg and Frits W. Vaandrager (Eds.). Lecture Notes in Computer Science, Vol. 1494. Springer-Verlag, London, 368–394. doi:10.1007/3-540-65193-4_29

[13] Ian Editor (Ed.). 2007. *The title of book one* (1st. ed.). The name of the series one, Vol. 9. University of Chicago Press, Chicago, Chapter The title of the chapter, 127–238. doi:10.1007/3-540-09237-4

[14] Ian Editor (Ed.). 2008. *The title of book two* (2nd. ed.). University of Chicago Press, Chicago, Chapter 100, 25–137. doi:10.1007/3-540-09237-4

[15] Matthew Van Gundy, Davide Balzarotti, and Giovanni Vigna. 2007. Catch me, if you can: Evading network signatures with web-based polymorphic worms. In *Proceedings of the first USENIX workshop on Offensive Technologies (WOOT '07)*. USENIX Association, Berkley, CA, Article 7, 9 pages.

[16] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. 1993. Maintaining Discrete Probability Distributions Optimally. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (Lecture Notes in Computer Science, Vol. 700)*. Springer-Verlag, Berlin, 253–264.

[17] David Harel. 1978. *LOGICS of Programs: AXIOMATICS and DESCRIPTIVE POWER*. MIT Research Lab Technical Report TR-200. Massachusetts Institute of Technology, Cambridge, MA.

[18] David Harel. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, Vol. 68. Springer-Verlag, New York, NY. doi:10.1007/3-540-09237-4

[19] Lars Hörmander. 1985. *The analysis of linear partial differential operators. III*. Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], Vol. 275. Springer-Verlag, Berlin, Germany. viii+525 pages. Pseudodifferential operators.

[20] Lars Hörmander. 1985. *The analysis of linear partial differential operators. IV*. Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], Vol. 275. Springer-Verlag, Berlin, Germany. vii+352 pages. Fourier integral operators.

[21] IEEE 2004. IEEE TCSC Executive Committee. In *Proceedings of the IEEE International Conference on Web Services (ICWS '04)*. IEEE Computer Society, Washington, DC, USA, 21–22. doi:10.1109/ICWS.2004.64

[22] Markus Kirschmer and John Voight. 2010. Algorithmic Enumeration of Ideal Classes for Quaternion Orders. *SIAM J. Comput.* 39, 5 (Jan. 2010), 1714–1747. doi:10.1137/080734467

[23] Donald E. Knuth. 1997. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd. ed.)*. Addison Wesley Longman Publishing Co., Inc., Boston.

[24] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY.

[25] Newton Lee. 2005. Interview with Bill Kinder: January 13, 2005. Video. *Comput. Entertain.* 3, 1, Article 4 (Jan.-March 2005). doi:10.1145/1057270.1057278

[26] Dave Novak. 2003. Solder man. Video. In *ACM SIGGRAPH 2003 Video Review on Animation theater Program: Part I - Vol. 145 (July 27–27, 2003)*. ACM Press, New York, NY, 4. doi:10.945/woot07-S422 http://video.google.com/videoplay?docid=6528042696351994555

[27] Barack Obama. 2008. A more perfect union. Video. Retrieved March 21, 2008 from http://video.google.com/videoplay?docid=6528042696351994555

[28] Poker-Edge.Com. 2006. Stats and Analysis. Retrieved June 7, 2006 from http://www.poker-edge.com/stats.php

[29] R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[30] Brian J. Reiser. 2014. Designing coherent storylines aligned with NGSS for the K-12 classroom. Presentation at National Science Education Leadership Association Meeting, Boston, MA, USA. https://www.academia.edu/6884962/

[31] Bernard Rous. 2008. The Enabling of Digital Libraries. *Digital Libraries* 12, 3, Article 5 (July 2008). To appear.

[32] Mehdi Saeedi, Morteza Saheb Zamani, and Mehdi Sedighi. 2010. A library-based synthesis methodology for reversible logic. *Microelectron. J.* 41, 4 (April 2010), 185–194.

[33] Mehdi Saeedi, Morteza Saheb Zamani, Mehdi Sedighi, and Zahra Sasanian. 2010. Synthesis of Reversible Circuit Using Cycle-Based Approach. *J. Emerg. Technol. Comput. Syst.* 6, 4 (Dec. 2010), 12 pages.

[34] Joseph Scientist. 2009. The fountain of youth. Patent No. 12345, Filed July 1st., 2008, Issued Aug. 9th., 2009.

[35] Stan W. Smith. 2010. An experiment in bibliographic mark-up: Parsing metadata for XML export. In *Proceedings of the 3rd. annual workshop on Librarians and Computers (LAC '10, Vol. 3)*, Reginald N. Smythe and Alexander Noble (Eds.). Paparazzi Press, Milan Italy, 422–431.

[36] Asad Z. Spector. 1990. Achieving application requirements. In *Distributed Systems* (2nd. ed.), Sape Mullender (Ed.). ACM Press, New York, NY, 19–33. doi:10.1145/90417.90738

[37] Harry Thornburg. 2001. *Introduction to Bayesian Statistics*. Retrieved March 2, 2005 from http://ccrma.stanford.edu/~jos/bayes/bayes.html, archived at [https://web.archive.org/web/20240505055615/https://ccrma.stanford.edu/~jos/bayes/bayes.html]

[38] TUG 2017. *Institutional members of the TEX Users Group*. Retrieved May 27, 2017 from http://wwtug.org/instmem.html

[39] Boris Veytsman. 2017. *acmart—Class for typesetting publications of ACM*. Retrieved May 27, 2017 from http://www.ctan.org/pkg/acmart

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009