MASTER'S THESIS 2026

AI- and Automation-Driven Workflow Optimization for Product Delivery

Zhe Zhang, Biyu Zou

ISSN 1650-2884
LU-CS-EX: 2026-20

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-20

AI- and Automation-Driven Workflow Optimization for Product Delivery

AI- och automationsdriven optimering av arbetsflöden för produktleverans

Zhe Zhang, Biyu Zou

AI- and Automation-Driven Workflow Optimization for Product Delivery

Zhe Zhang
zh3565zh-s@student.lu.se

Biyu Zou
bi8351zo-s@student.lu.se

June 4, 2026

Master's thesis work carried out at Ericsson AB.

Supervisors: Anne Sjögren, anne.sjogren@ericsson.com
Lifeng Han, lifeng.han@ericsson.com
Pierre Nugues, pierre.nugues@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Modern telecommunications research and development (R&D) rely on a complex ecosystem of enterprise systems for documentation, planning, execution tracking, and managerial reporting. To manage this scale, massive requirements are typically broken down into bounded sub-projects or feature modules, known internally at Ericsson as Solution Packages (SPs). In this SP workflow, critical project knowledge is distributed across distinct platforms—primarily Confluence, Jira, and Feature & Product Tracker (FPT). Consequently, project coordinators, referred to as SP Drivers, face a severe information fragmentation problem, requiring manual interpretation of status, evidence, risks, and task priorities across multiple disconnected environments.

This thesis presents the design, implementation, and architectural analysis of **SP Copilot**, presented through the SP Guardian interface, an AI-assisted multi-skill prototype tailored for project coordination. To address the latency, traceability, and deterministic requirements of enterprise settings, the system deliberately departs from conventional vector-based retrieval-augmented generation (RAG) in its core summary and risk workflows. Instead, it adopts a hybrid architecture that integrates exact glossary indexing for robust terminology grounding, deterministic rule-based heuristics, and constrained large language model (LLM) synthesis.

Furthermore, the prototype introduces a decoupled integration layer based on the Model Context Protocol (MCP). By delegating low-level system interactions with Confluence, Jira, and FPT to local JavaScript MCP servers, the high-level Python layer maintains focus on orchestration, skill routing, prompt assembly, and workflow control. The copilot demonstrates capabilities across five functional dimensions: multi-source context summarization, human-in-the-loop task synchronization, risk-oriented diagnosis, rule-assisted Jira task prioritization, and offline local knowledge-base lookup. Ultimately, this work contributes an architectural blueprint for enterprise AI copilots that prioritize evidence grounding, reduce manual synchronization overhead, and enhance human decision-making without compromising operational governance.

Keywords: workflow automation, large language models, Model Context Protocol, enterprise systems, project intelligence, human-in-the-loop, telecommunications

Acknowledgements

This thesis has been shaped not only by research questions, architectural decisions, and lines of code, but also by the people who accompanied us through the process.

We would like to express our deepest gratitude to our supervisors at Ericsson, Anne Sjögren and Lifeng Han. Their guidance opened a window into the practical realities of industrial work, where technology, coordination, responsibility, and human judgment meet every day. Through their patience, insight, and trust, this project gradually found both its direction and its grounding in a real organizational context.

We are sincerely grateful to our academic supervisor, Pierre Nugues, for his thoughtful advice, careful reading, and steady encouragement. His feedback helped us see the work not only as a prototype, but as a research contribution with a clearer structure, stronger argument, and wider relevance.

We would also like to thank the colleagues at Ericsson who generously shared their time, experience, and domain knowledge with us. Their perspectives helped us understand that enterprise systems are never only technical infrastructures; they are also traces of collaboration, decisions, constraints, and everyday work.

Finally, we thank our families and friends for their patience, warmth, and quiet support throughout this journey. Their encouragement gave us a place to return to when the work felt uncertain or demanding. We are also grateful to each other, for the long discussions, shared persistence, and mutual trust that carried this thesis from its first vague ideas to its final pages.

Contents

Chapter 1
Introduction

The development of modern telecommunications products is not only a technical engineering activity, but also a large-scale coordination problem. Product delivery in industrial research and development (R&D) settings requires continuous alignment among planning targets, implementation activities, verification milestones, evidence collection, and managerial reporting. In large organizations such as Ericsson, these activities are distributed across multiple enterprise systems, each designed to support a specific operational perspective. While such specialization improves local efficiency, it also generates a fragmented information landscape in which no single tool captures the full state of a project. Similar coordination challenges have been widely discussed in project management, software architecture, DevOps, business process management, and AI-assisted project management literature (Project Management Institute, 2021; Kerzner, 2017; Bass et al., 2015; Dumas et al., 2018; Weske, 2019; Auth et al., 2019).

This thesis addresses that challenge in the context of Ericsson's *Solution Package* (SP) workflow. To contextualize this within standard software engineering, an SP can be understood as a bounded sub-project or a large feature module. It acts as a structured delivery unit used to organize, follow up, and govern product or feature development toward predefined milestones. In practice, an SP serves as a coordination entity that connects technical implementation, project planning, checklist compliance, dependencies, and milestone readiness. Managing an SP therefore requires the continuous synthesis of information across documentation systems, task trackers, and planning tools. This synthesis is time-consuming and cognitively demanding, especially when users must manually cross-reference incomplete or partially inconsistent data.

The practical consequence is that project coordinators spend substantial effort not only on making decisions, but on *assembling the context required to make those decisions.* This context assembly includes reading long documentation pages, checking the current state of linked Jira issues, examining milestone dates, validating whether evidence is present, and interpreting domain-specific terminology. Such work is both repetitive and error-prone. When data are dispersed across disconnected systems, even experienced coordinators may struggle to form

a reliable and timely overview of project health.

To address these issues, this thesis investigates the design of an AI-assisted enterprise workflow support system, called **SP Copilot** and presented to users through a frontend referred to as *SP Guardian*. Rather than pursuing a fully autonomous agent, which would be difficult to justify in a governance-heavy industrial environment, the proposed artifact acts as an intelligent copilot. Its purpose is to assist, not replace, human coordinators by integrating data across systems, grounding terminology, generating evidence-based summaries, supporting risk-oriented reasoning, and proposing controlled synchronization actions subject to explicit human approval. This design direction aligns with broader trends in human-in-the-loop AI, human-centered AI, explainable AI, and assistive project-management systems (Auth et al., 2019; de Bem et al., 2024; Holzinger, 2016; Shneiderman, 2020; Adadi and Berrada, 2018).

1.1 Industrial Background and Domain Context

The research is conducted in collaboration with a division within Ericsson's RCE organization, responsible for driving Solution Packages as part of telecommunications product and feature development. The organizational motivation behind the project is to improve cross-functional development efficiency through automation, data integration, and AI-assisted decision support. In this setting, project coordination includes milestone tracking, evidence follow-up, dependency monitoring, task prioritization, and readiness communication across teams with different responsibilities and system views.

The primary user targeted by the proposed system is the *SP Driver*. This role is highly coordination-intensive. An SP Driver is expected to understand milestone expectations, monitor implementation progress, ensure that evidence is collected and documented, detect emerging risks, and support the communication of project state to stakeholders. Although these tasks are central to successful delivery, much of the work is administrative in nature and depends on the ability to gather information from multiple sources quickly and accurately. The growing interest in AI-assisted project coordination and decision support reflects the importance of reducing this overhead (Auth et al., 2019; de Bem et al., 2024).

To make the thesis understandable beyond its industrial setting, the most important enterprise systems and domain concepts are introduced below. A more detailed account of the data and knowledge sources is provided later in Chapter 3.

Confluence Confluence is the primary enterprise documentation and collaboration platform used in the SP workflow. It contains project pages, milestone descriptions, readiness evidence, follow-up comments, and references to related material. Within the SP process, Confluence functions not only as a wiki but also as a documentation and compliance surface where project state is recorded over time.

Jira Jira is the enterprise issue and task tracking system used to manage operational execution. It contains work items such as epics (large-scale initiatives encapsulating multiple smaller tasks), stories, tasks, sub-tasks, assignees, due dates, workflow states, priorities, and

comments. Compared with Confluence, Jira provides a more granular view of current execution, but it often lacks the broader milestone and planning context needed for managerial interpretation.

Feature & Product Tracker Feature & Product Tracker (FPT) provides product-level planning and scheduling context. It may contain planning-related metadata such as gate dates, release associations, backlog views, status phases, and broader scheduling information. In the SP workflow, FPT complements Confluence and Jira by connecting local project execution to formal product-planning structures.

The F-Checklist The F-Checklist is a domain-specific artifact embedded in Confluence and used to assess milestone readiness. It contains criteria, evidence expectations, comments, and status indicators required to evaluate whether a project is ready to pass a particular development gate. Its practical importance is high because it connects day-to-day project execution with formal milestone governance.

SP Driver The SP Driver is the project coordination role responsible for driving and monitoring the progress of a Solution Package. This role involves following up readiness criteria, communicating milestone state, checking evidence completeness, aligning planning and execution, and helping stakeholders understand emerging concerns. Because the role depends heavily on cross-system interpretation and synthesis, it is particularly well suited for support through intelligent automation and AI-assisted context building.

1.2 Problem Statement

The central problem addressed in this thesis is the fragmentation of SP project knowledge across multiple enterprise systems. As illustrated in Figure 1.1, documentation, execution tracking, planning information, readiness checklists, and procedural knowledge are distributed across separate sources. Each source provides a useful but partial view of the project. The coordination burden arises because the SP Driver must manually combine these partial views into a coherent understanding of project state, risk, and next actions.

This fragmentation introduces several operational bottlenecks:

1. **Manual data synthesis:** Understanding project state requires extensive manual data synthesis. Project signals are distributed across long documentation pages, issue trackers, status fields, comments, planning views, and milestone tables. Users must manually navigate between systems and mentally integrate what they find. This is inefficient and introduces the risk that important details are overlooked.

2. **Consistency maintenance:** Maintaining consistency between documentation and execution systems requires repetitive and error-prone manual work. Checklist items may need to be mirrored as Jira tasks, or Jira progress may need to be reflected back into Confluence reporting structures. Without targeted automation, this leads to duplicated effort and stale information. More generally, research on software delivery, traceability, and project tooling has long recognized that fragmented lifecycle tools create

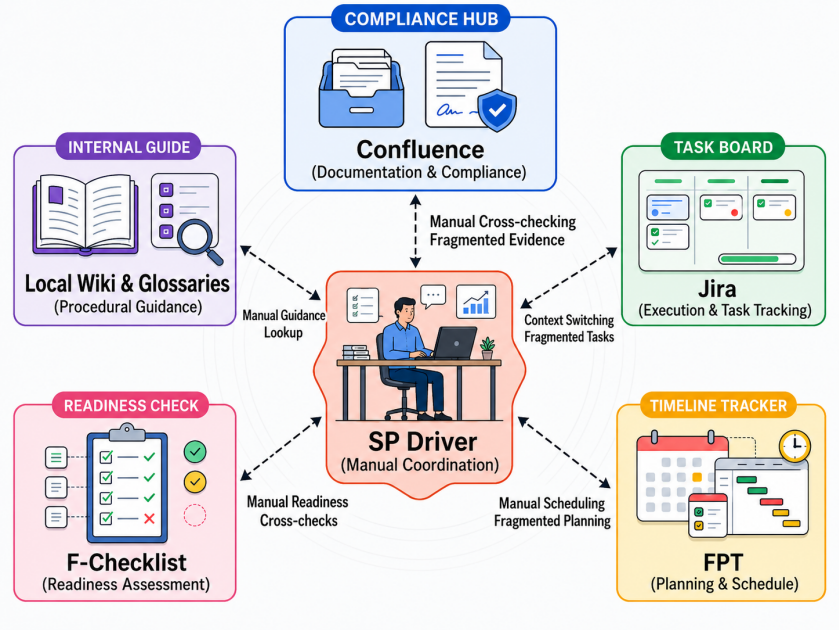


Figure 1.1: Information fragmentation in the SP workflow across documentation, execution, planning, checklist, and knowledge-base sources.

synchronization and traceability burdens (Bass et al., 2015; Gotel and Finkelstein, 1994; Cleland-Huang et al., 2014; Mäder and Egyed, 2012).

3. **Risk and priority interpretation:** The interpretation of risk and task priority remains difficult to scale. In industrial environments, project concerns are rarely expressed as a single explicit label. Rather, they emerge as weak signals: missing evidence, outdated comments, unresolved blockers, terminology ambiguity, or inconsistencies between what is documented and what is actually tracked. General-purpose large language models can struggle in this environment because they may hallucinate data, misread abbreviations, or overgeneralize from incomplete context. The importance of identifying such risks early has long been emphasized in software risk management (Boehm, 1991; Wallace et al., 2004), while recent work on LLMs has highlighted both their promise and their reliability limitations in technical settings (Hou et al., 2024; Huang et al., 2024; Ji et al., 2023).

These challenges motivate the central research problem of the thesis:

> *How can a hybrid AI and automation architecture assist project coordinators in mitigating information fragmentation, ensuring cross-system synchronization, and supporting evidence-based risk and priority diagnosis across disparate enterprise platforms?*

1.3 Research Objectives and Scope

The objective of the thesis is to design, instantiate, and analyze a multi-skill enterprise copilot that supports project coordination in the SP workflow. The artifact is intended to augment

the work of the SP Driver by improving context understanding, reducing manual overhead, and providing more grounded support for decision making.

The prototype realizes five functional capabilities:

1. **Multi-source context summarization:** synthesize live data from Confluence, Jira, and FPT to generate concise, factual project overviews.
2. **Human-in-the-loop synchronization:** detect state discrepancies between documentation and execution trackers and propose update actions subject to explicit user approval.
3. **Risk-oriented diagnosis:** combine deterministic heuristics with constrained LLM synthesis to evaluate milestone evidence and surface grounded project concerns.
4. **Rule-assisted prioritization:** analyze Jira issue context against project milestones and planning constraints to recommend read-only prioritizations for SP Driver triage.
5. **Offline knowledge-base lookup:** retrieve and synthesize internal operational guidance from a local Markdown repository using lightweight retrieval methods.

The scope of the thesis is the architectural design, implementation, and evaluation of this workflow-support artifact. The work includes data integration, prompt engineering, terminology grounding, skill routing, synchronization logic, evidence preparation, and user-facing copilot behavior. The project emphasizes practical enterprise constraints such as traceability, deterministic grounding, latency, permissions, and governance compliance.

The thesis does not aim to develop a fully autonomous project-management agent. Confluence, Jira, and FPT remain the authoritative enterprise systems, and the SP Copilot operates as a support layer that helps users interpret, compare, and act upon information more effectively. In particular, write-capable actions are constrained by human-in-the-loop approval rather than delegated to the language model.

The thesis also does not train a standalone predictive machine-learning model for statistical delay forecasting. Earlier project ideas considered stronger machine-learning-based prediction, but available historical data were not sufficient to support robust and generalizable statistical modeling. Consequently, the final design prioritizes evidence-based interpretation and grounded synthesis over unsupported numerical prediction. This shift is consistent with broader concerns in explainable and trustworthy AI, where actionable support often matters more than opaque scores (Guidotti et al., 2019; Adadi and Berrada, 2018; Doshi-Velez and Kim, 2017).

1.4 Thesis Contributions

The thesis makes the following primary contributions:

1. **A hybrid enterprise copilot architecture** that combines Python-based orchestration with a Model Context Protocol (MCP) integration layer for enterprise system access.
2. **A deterministic terminology-grounding strategy** that replaces vector-based glossary retrieval with exact, tiered glossary indexing to improve correctness and latency.

3. **A human-in-the-loop synchronization design** that supports safe and explainable alignment between Confluence checklist structures and Jira execution artifacts.

4. **A rule-assisted AI diagnostic workflow** that integrates deterministic expert heuristics with constrained LLM synthesis for project risk and prioritization support.

5. **An architectural case study of enterprise AI deployment** that highlights how practical requirements such as governance, fallback resilience, and evidence traceability influence system design.

1.5 Use of Generative AI

Generative AI tools were used during the preparation of this thesis as writing and formatting support. In particular, they were used to improve grammar, clarity, and expression; to assist with LaTeX formatting and restructuring; and to support the Swedish translation and language refinement of the popular-science summary.

Generative AI was not treated as an authoritative source of technical or empirical claims. All architectural descriptions, implementation details, evaluation results, interpretations, and final wording were reviewed, selected, and edited by the authors. The responsibility for the content of the thesis, including the correctness of the technical work and the conclusions drawn from it, remains with the authors.

In addition to writing assistance, the project itself studies and implements generative-AI-based functionality as part of the SP Copilot prototype. The use of LLMs inside the system and in the generated-output quality evaluation is described explicitly in the methodology, implementation, and evaluation chapters.

1.6 Author Contributions

This thesis was carried out jointly by Zhe Zhang and Biyu Zou. The overall system idea, architectural direction, and initial prototype framework were developed collaboratively. Both authors contributed to the design of the multi-skill copilot architecture, the integration of LLM-based workflows, and the refinement of the thesis argument.

Zhe Zhang was primarily responsible for the MCP-based implementation, including connecting the prototype to enterprise data platforms through MCP servers and implementing low-level tool access. Zhe also contributed substantially to the skill logic implementation, system deployment, evaluation design and execution, and the writing and revision of the thesis.

Biyu Zou was primarily responsible for orchestration-related logic, the skill implementation, the local data and knowledge-base components, and earlier retrieval-based prototype work, including the legacy RAG implementation. Biyu also contributed substantially to the overall architecture, implementation refinement, testing, and thesis development.

Although responsibilities were divided in this way, the project was iterative and collaborative. Design decisions, debugging, evaluation interpretation, and final thesis integration were discussed and refined jointly.

1.7 Thesis Structure

The remainder of the thesis is organized as follows. Chapter 2 introduces the technical background and related work needed to understand and position the SP Copilot, including enterprise product development, AI-assisted project coordination, large language models, grounding strategies, MCP-based tool integration, and human-in-the-loop governance. Chapter 3 describes the operational data sources, glossary indexes, and offline knowledge repositories used by the system. Chapter 4 presents the research method and the system's overall architecture. Chapter 5 details the implementation of the five copilot skills. Chapter 6 presents the evaluation design, results, and analysis of router accuracy, generated output quality, and operational behavior. Chapters 7 and 8 discuss the implications of the work and conclude the thesis. Formal abstractions of the core workflows are provided in Appendix A, and the main prompt-template design is documented in Appendix B.

Chapter 2
Background and Related Work

This chapter combines the technical background and related-work discussion needed to position the SP Copilot. The purpose is not to provide a general survey of all work on large language models, retrieval, workflow automation, or project management. Instead, the chapter introduces the concepts that are necessary to understand the system and then relates them to the specific research gap addressed by the thesis.

The chapter is organized around five themes. First, it discusses enterprise product development and AI-assisted project coordination. Second, it introduces large language models as support tools for enterprise knowledge work and project coordination. Third, it examines grounding strategies, including retrieval-augmented generation and exact indexing. Fourth, it discusses tool-using language models and MCP-based integration. Fifth, it reviews human-in-the-loop governance, explainability, and trust. The final section synthesizes these areas into the research gap addressed by the SP Copilot.

2.1 Enterprise Product Development and AI-Assisted Project Coordination

Large-scale enterprise product development, especially in telecommunications R&D, is a coordination-intensive activity. Product delivery involves requirements interpretation, planning, implementation tracking, verification, evidence collection, and reporting. These activities are often distributed across different enterprise lifecycle-management and collaboration tools. Requirements or milestone descriptions may be documented in one platform, execution tracked in another, and planning data maintained in a third. This fragmentation has been recognized as a source of duplicated manual effort, weak traceability, and inconsistent project state across tools (Bass et al., 2015; Gotel and Finkelstein, 1994; Cleland-Huang et al., 2014; Mäder and Egyed, 2012).

Classical project-management literature emphasizes structured planning, monitoring,

key performance indicators, risk management, and stakeholder communication (Project Management Institute, 2021; Kerzner, 2017). Business process management and DevOps research similarly highlight the importance of process visibility, feedback loops, automation, and cross-functional coordination (Dumas et al., 2018; Weske, 2019; Bass et al., 2015). In practice, however, process visibility depends on whether users can interpret heterogeneous evidence across several systems rather than only inspect a single dashboard.

AI-assisted project-management research has explored automated status tracking, schedule prediction, risk identification, resource allocation, and decision-support dashboards (Auth et al., 2019; de Bem et al., 2024). These directions are relevant to the SP workflow because they address the overhead of interpreting project state. However, many coordination problems are not purely predictive. An SP Driver often needs to understand whether evidence is missing, whether Jira execution is aligned with checklist documentation, whether comments indicate blockers, and whether planning dates make some tasks more urgent. A numerical prediction alone is not sufficient if the supporting evidence cannot be inspected.

The SP Copilot is therefore positioned as an assistive coordination system rather than as a predictive project-management model. Its goal is to reduce the effort required to assemble and interpret project context across Confluence, Jira, FPT, checklist structures, glossary files, and local wiki documents. This emphasis on evidence interpretation shapes the system's architecture: the prototype prioritizes multi-source data access, traceable summaries, human-approved synchronization, and rule-assisted diagnosis over fully autonomous project control.

2.2 Large Language Models for Enterprise Workflow Support

Large language models (LLMs) are neural language models trained on large text corpora and adapted to generate, summarize, transform, and reason over natural language. Modern LLMs are closely connected to the Transformer architecture, which introduced an attention-based mechanism for modeling relationships among tokens in a sequence (Vaswani et al., 2017). Unlike earlier sequence models that processed text primarily in order, Transformers can represent dependencies across a broader context window more efficiently, which has made them foundational for many recent language technologies.

While the original Transformer architecture featured both an encoder and a decoder, the majority of modern LLMs—including the GPT family—rely on a *decoder-only* architecture. In a decoder-only model, the network generates text autoregressively, predicting the next token based exclusively on the sequence of preceding tokens. This approach, combined with massive scale, has proven highly effective for open-ended generation, few-shot prompting, and reasoning tasks (Radford et al., 2019; Brown et al., 2020).

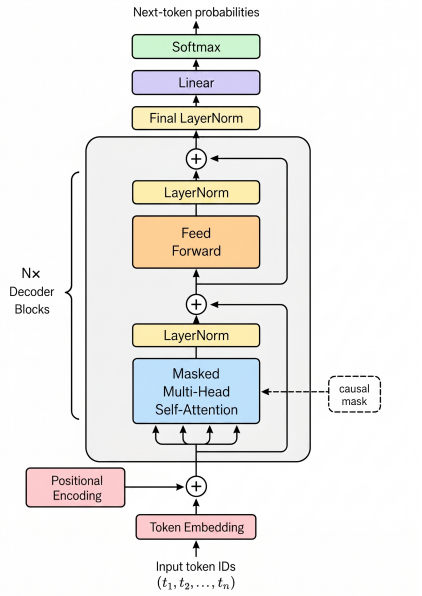


Figure 2.1: Simplified illustration of a decoder-only Transformer architecture, adapted from Vaswani et al. (2017).

As illustrated in Figure 2.1, the decoder-only architecture is built around stacked masked self-attention and feed-forward layers. This thesis does not propose modifications to the underlying Transformer architecture. Instead, it utilizes commercial decoder-only LLMs as reasoning and text-synthesis engines inside an enterprise workflow-support system. Pretrained language models can be further adapted through instruction tuning and alignment methods so that they respond more effectively to user tasks (Ouyang et al., 2022).

Across software engineering and enterprise knowledge-work contexts, LLMs have been applied to code generation, documentation, issue summarization, bug analysis, requirements support, and developer assistance (Hou et al., 2024; Fan et al., 2023). These capabilities also make LLMs attractive for enterprise project coordination, where users frequently need to summarize long documents, compare textual records, interpret comments, and produce concise status reports for stakeholders.

Prompting is central to such systems because the model's behavior is shaped by the instructions and evidence supplied at runtime. Prompt-engineering research has studied patterns such as in-context learning and chain-of-thought prompting (Brown et al., 2020; Wei et al., 2022; Kojima et al., 2022). In the SP Copilot, prompting is used in a more conservative operational role. Runtime prompts are assembled from tagged evidence blocks such as Confluence page content, Jira issue data, FPT planning context, checklist evidence, glossary entries, and local wiki excerpts. The prompts also specify constrained output formats such as simple HTML or strict JSON. This makes outputs easier to render, test, and debug.

At the same time, LLM-based systems introduce reliability risks. Hallucination and faithfulness failures are well-known challenges in natural-language generation and LLM-based support systems (Ji et al., 2023; Huang et al., 2024; Maynez et al., 2020). In an enterprise workflow, unsupported statements may have practical consequences: a model must not invent issue states, milestone outcomes, missing evidence, or completed actions. The SP Copilot therefore treats the LLM as a synthesis component rather than an authority. Source systems remain authoritative, and the language model is constrained by retrieved evidence, deter-

ministic preprocessing, explicit task prompts, and application-level governance.

2.3 Grounding Strategies: RAG, Retrieval, and Exact Indexing

Retrieval-augmented generation (RAG) is a common architecture for grounding language-model outputs in external information. A RAG system retrieves relevant documents or passages and supplies them to the model as context at generation time (Lewis et al., 2020; Gao et al., 2024). Dense retrieval methods represent queries and documents as vectors in an embedding space, allowing semantically related texts to be found even when they do not share exact words (Karpukhin et al., 2020; Reimers and Gurevych, 2019). Approximate nearest-neighbor algorithms make large-scale vector search more practical (Johnson et al., 2019; Malkov and Yashunin, 2018).

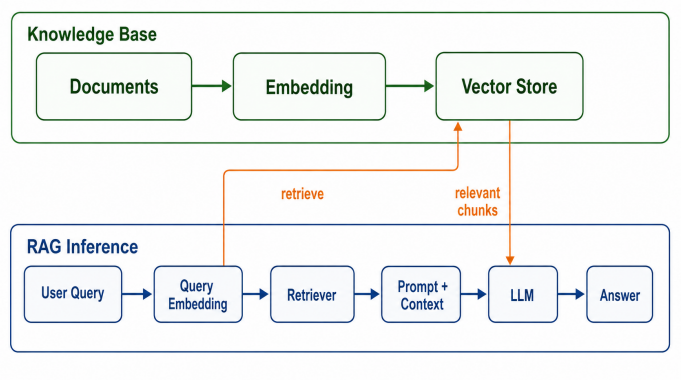


Figure 2.2: Typical retrieval-augmented generation architecture, where documents are embedded into a vector store and relevant chunks are retrieved as context for language-model generation.

As illustrated in Figure 2.2, a typical RAG workflow consists of two connected processes. In the indexing process, documents are transformed into embeddings and stored in a vector database. At inference time, the user query is also embedded, relevant chunks are retrieved, and the retrieved context is combined with the prompt before being sent to the language model. This general grounding pattern is useful for enterprise question answering over longer documents. In the SP Copilot, the local wiki lookup skill follows the same principle of supplying retrieved context to the LLM, but it uses lightweight lexical retrieval rather than a full vector-store pipeline.

However, retrieval is not a single technique. Classical information-retrieval methods such as term weighting, inverse document frequency, BM25, and sparse retrieval remain important when exact identifiers, short phrases, or domain-specific terms are central to the task (Spärck Jones, 1972; Salton and Buckley, 1988; Robertson and Zaragoza, 2009; Manning et al., 2008; Baeza-Yates and Ribeiro-Neto, 2011; Lin et al., 2021). This distinction matters in acronym-heavy enterprise environments. A short internal abbreviation may have a precise meaning within one workflow, while semantic vector search may retrieve a plausible but incorrect definition.

The SP Copilot therefore uses a mixed grounding strategy. For procedural documents and longer local wiki entries, lightweight retrieval is useful. For core terminology grounding in summary and risk workflows, the system uses exact, tiered glossary indexing rather than broad semantic similarity search. This design is narrower than general RAG, but it better matches the precision, latency, and inspectability requirements of the SP workflow.

2.4 Tool-Using Language Models and MCP-Based Integration

Many AI applications need access to external tools, databases, or APIs. Tool-using language-model research investigates how models can reason about when to call tools and how to incorporate tool results into later answers. ReAct combines reasoning and acting in language-model workflows (Yao et al., 2023); Toolformer explores how models can learn to use tools (Schick et al., 2023); WebGPT and Gorilla show related patterns for connecting language models to browsing or API-based capabilities (Nakano et al., 2021; Patil et al., 2023). Broader surveys describe tool learning as an important direction for foundation-model systems (Qin et al., 2023).

Enterprise copilots often need this kind of tool access because useful context is stored in live systems rather than in static document collections. In the SP workflow, the copilot needs information from Confluence pages, Jira issues, FPT planning records, checklist structures, local glossary files, and offline knowledge repositories. Directly embedding all integration logic inside prompts or model calls would be difficult to maintain and hard to govern.

The Model Context Protocol (MCP) is a recent standardization effort for connecting AI applications to external tools and data sources through a common protocol (Anthropic, 2024; Model Context Protocol, 2025b,a). Its general architecture follows a client-server pattern, where an AI application connects to one or more MCP servers that expose external tools, data, or resources.

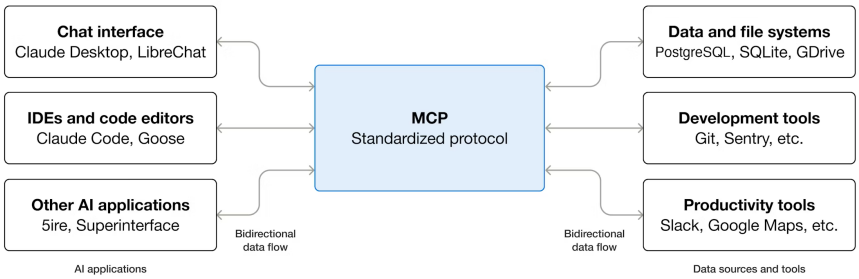


Figure 2.3: General architecture of the Model Context Protocol. Adapted from Anthropic's MCP materials (Anthropic, 2024).

In the SP Copilot, MCP is used in a conservative engineering role. The language model is not given open-ended control over enterprise tools. Instead, local JavaScript MCP servers encapsulate low-level interactions with systems such as Confluence, Jira, and FPT, while the

Python layer controls orchestration, skill routing, evidence preparation, prompt assembly, and workflow boundaries. This preserves a clear separation between language synthesis and enterprise-system operations. It also supports governance, because write-capable actions can be guarded by application logic and explicit user approval rather than delegated to unconstrained model behavior.

2.5 Human-in-the-Loop Governance, Explainability, and Trust

Human-in-the-loop automation refers to systems in which automated components assist with analysis or action proposals while humans remain involved in validation, judgment, or execution. This is especially important when automated actions may affect operational records, accountability, or downstream decisions. Interactive machine learning and human-centered AI research emphasize that human oversight is often necessary in high-impact settings (Amershi et al., 2014; Holzinger, 2016; Shneiderman, 2020).

Trust in automation is not simply a matter of increasing autonomy. Users need appropriate reliance: they should trust the system when it is reliable, but remain aware of its limitations (Lee and See, 2004). Research on levels of automation and situation awareness similarly warns that excessive automation can reduce human control and understanding (Parasuraman et al., 2000; Endsley, 1995). For an enterprise copilot, this is especially relevant because fluent generated text may make unsupported claims appear more authoritative than they are.

Explainable AI focuses on making system behavior understandable to users. Surveys of XAI methods emphasize that explanations are important when users need to evaluate, trust, debug, or contest system outputs (Adadi and Berrada, 2018; Guidotti et al., 2019; Doshi-Velez and Kim, 2017). Although local explanation methods such as LIME and SHAP were developed mainly for predictive models (Ribeiro et al., 2016; Lundberg and Lee, 2017), the underlying principle also applies to LLM-based decision support: users need to know what evidence an output is based on and where its limitations are.

Enterprise AI also depends on organizational knowledge. Knowledge-management theory distinguishes between tacit knowledge held by people and explicit knowledge captured in documents, processes, and repositories (Nonaka and Takeuchi, 1995; Davenport and Prusak, 1998). In the SP workflow, important meanings are stored in Confluence pages, checklist structures, glossary files, and local wiki documents. The SP Copilot therefore treats terminology grounding and local knowledge retrieval as architectural components rather than optional documentation aids.

These considerations explain why the SP Copilot is designed as a governed copilot rather than as a fully autonomous agent. It supports users by gathering and synthesizing evidence, detecting inconsistencies, diagnosing risks, and proposing synchronization actions. However, write-capable actions remain human-approved, recommendations are read-only unless explicitly confirmed, and generated outputs are grounded in source evidence. The system is intended to reduce coordination overhead while preserving SP Driver responsibility for consequential decisions.

2.6 Research Gap and Positioning

Existing research provides strong foundations in workflow automation, project-management AI, LLM-based enterprise support, retrieval-augmented generation, tool-using language models, explainability, and human-in-the-loop automation. However, these areas often address separate parts of the problem. Traditional workflow integration can synchronize structured systems, but it often assumes cleaner schemas and more stable artifact relationships than are available in semi-structured enterprise documentation. AI-assisted project-management work often emphasizes prediction or dashboards, while the SP workflow requires evidence interpretation, terminology grounding, and support for human coordination. LLM-based support tools show the usefulness of generative models, but they also highlight the risks of hallucination and unsupported inference.

The research gap addressed by this thesis is therefore practical as well as technical. Few approaches simultaneously address all of the following requirements in one concrete enterprise copilot architecture:

- fragmented project knowledge across documentation, execution, planning, checklist, and wiki sources;
- deterministic terminology grounding for acronym-heavy enterprise environments;
- protocol-based enterprise integration with conservative orchestration control;
- human-approved synchronization between documentation and issue-tracking systems;
- evidence-based summary, risk diagnosis, prioritization, and knowledge lookup in one multi-skill workflow.

The SP Copilot addresses this gap by integrating multi-source data access, exact glossary indexing, rule-based diagnostics, constrained LLM synthesis, and human-in-the-loop action control into a single artifact tailored to the SP workflow. This combination is important because the target problem is not only to answer questions, but to support a governed coordination workflow across fragmented enterprise systems.

2.7 Summary

This chapter introduced the technical and research foundations needed to understand the SP Copilot. Enterprise workflow coordination and AI-assisted project-management research explain the operational problem: SP Drivers must interpret fragmented evidence across several systems. LLM-based enterprise support research explains why generative models are useful for summarization and synthesis, while also highlighting the need to control hallucination and unsupported inference. Retrieval and grounding research shows that different knowledge types require different retrieval strategies, motivating the system's use of exact glossary indexing alongside lightweight local wiki retrieval.

The chapter also positioned MCP-based tool integration and human-in-the-loop governance as architectural requirements rather than peripheral implementation details. MCP supports decoupled access to enterprise tools, but the SP Copilot uses it conservatively through

Python-controlled orchestration. Human-in-the-loop design, explainability, and trust research justify the system's evidence-grounded outputs, read-only recommendations, and explicit approval boundaries for write-capable actions.

Together, these areas define the thesis contribution: a governed, evidence-grounded, multi-skill enterprise copilot for coordination-heavy SP workflows. The following chapters build on this positioning by describing the data and knowledge sources used by the system, the architecture of the prototype, its implementation, and its evaluation.

Chapter 3

Data and Knowledge Sources

Unlike traditional machine learning studies that rely on a fixed, static dataset, the SP Copilot operates over a dynamic enterprise information space. The system does not consume one preassembled table of observations. Instead, it continuously interprets live system states, structured records, local indexes, and offline knowledge assets. For this reason, the term *dataset* is somewhat insufficient; a more accurate description is that the system is built on a *multi-source operational information environment*.

This chapter describes the different classes of data used by the copilot, how they differ in structure and role, and what constraints they impose on system design.

3.1 Primary Operational Data Sources

The operational workflow relies on API-driven extraction from three main enterprise environments: Confluence, Jira, and Feature & Product Tracker (FPT). Each source contributes a different view of the SP workflow. Confluence captures documentation and compliance evidence, Jira captures execution state, and FPT contributes planning and product-level scheduling context. The purpose of the copilot is not to replace these systems, but to combine their signals into a more coherent coordination view.

A central challenge is that none of these systems is complete on its own. Confluence may describe milestone expectations but not reflect the latest execution state. Jira may show task progress but not explain why a task matters for a specific gate. FPT may contain planning commitments but not the detailed evidence behind them. The SP Copilot therefore treats the three systems as complementary sources whose signals must be cleaned, normalized, and interpreted together.

3.1.1 Confluence

Confluence provides the documentation and compliance layer of the SP workflow. It contains page body text, milestone descriptions, checklist matrices, evidence comments, linked references, and macro-generated sections. In the SP context, this makes Confluence the most narrative-rich source, but also the most heterogeneous one. A single page may contain current execution updates, historical background, generic process guidance, checklist rows, embedded tables, and links to external evidence.

The copilot therefore treats Confluence as a mixed structured–unstructured data source. Some information can be extracted deterministically from tables or checklist rows, while other information must be cleaned from page text, comments, and embedded page structures. Before the content is used in prompts or synchronization logic, the system separates page text, checklist data, milestone activity rows, comments, and linked references into distinct evidence blocks.

This separation is important because different workflows depend on different parts of the page. The summary skill needs a broad but compact project overview. The risk skill needs evidence about open concerns, missing items, and gate readiness. The synchronization workflow needs task-like structures that can be compared against Jira issues. Treating the Confluence page as one undifferentiated text block would make these tasks less reliable and more difficult to inspect.

Figure 3.1 illustrates this extraction process. The key implementation issue is that Confluence pages often combine formal checklist evidence with free-text updates and historical notes. This heterogeneity is one reason why deterministic parsing and careful preprocessing are necessary before the content can be used for summary generation, risk diagnosis, or synchronization proposals.

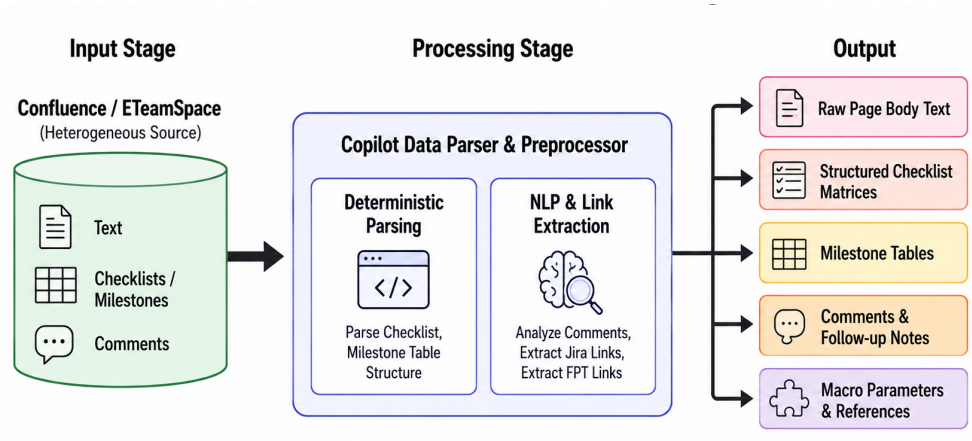


Figure 3.1: Data extraction and categorization flow from a heterogeneous Confluence page.

3.1.2 Jira

Jira provides the execution-oriented view of project state. While Confluence describes milestone expectations and documented evidence, Jira records operational work through issues, workflow states, assignees, priorities, due dates, comments, and dependency links. In the

SP Copilot, Jira therefore acts as the main source for understanding what work is currently open, assigned, blocked, recently updated, or linked to other work items.

Compared with Confluence, Jira fields are more structured and better suited to deterministic extraction. The system can retrieve issue keys, summaries, workflow states, priorities, assignees, update timestamps, linked issues, and comments. These fields are then normalized into compact evidence blocks that can support summaries, risk interpretation, synchronization checks, and prioritization recommendations.

However, structure does not automatically mean interpretation. A Jira issue in backlog, in progress, or blocked state only becomes meaningful when interpreted relative to the current SP milestone, expected evidence, and planning commitments. For example, an unresolved issue may be harmless if it belongs to a later phase, but critical if it is tied to an upcoming F-gate or missing readiness evidence. This is why the copilot does not treat Jira as a standalone project-health source.

Figure 3.2 shows how Jira contributes structured execution signals. However, Jira alone does not provide sufficient managerial interpretation. A task may be marked as ongoing or blocked, but the significance of that state depends on milestone timing, evidence expectations, Confluence documentation, and FPT planning context. For this reason, Jira is treated as an execution signal source rather than as a complete project-health model.

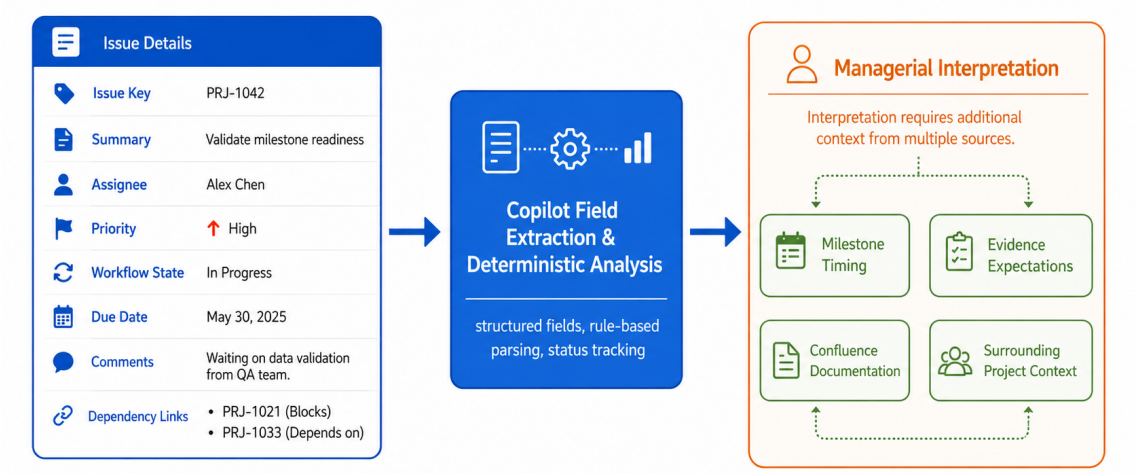


Figure 3.2: Jira-based execution state extraction for capturing structured issue signals and supporting cross-source managerial interpretation.

3.1.3 Feature & Product Tracker

Feature & Product Tracker contributes planning and organizational scheduling context. Depending on the scenario and access scope, the system may retrieve backlog item metadata, status phases, target releases, gate dates, linked planning identifiers, ownership information, and product-level notes. In practice, FPT provides a higher-level planning view that complements the documentation focus of Confluence and the execution focus of Jira.

This role is especially important because local execution signals can be misleading when interpreted in isolation. A Jira issue may appear operationally minor, but become important when connected to an upcoming milestone, target release, or gate expectation. Conversely,

a missing or unavailable FPT field may itself represent a planning-evidence gap when the project is preparing for a milestone decision.

FPT also helps anchor the copilot's interpretation to formal planning structures. Whereas Confluence may contain narrative explanations and Jira may contain task-level execution data, FPT provides information closer to the product-planning layer. This makes it useful for interpreting whether a project has sufficient planning evidence, whether schedule information is available, and whether local execution work can be connected to broader delivery commitments.

Figure 3.3 illustrates how FPT helps connect local project evidence to broader product-level planning. In the SP Copilot, FPT context is particularly relevant for summary and prioritization workflows, where the system must interpret current execution status against milestone timing, target releases, and planning commitments.

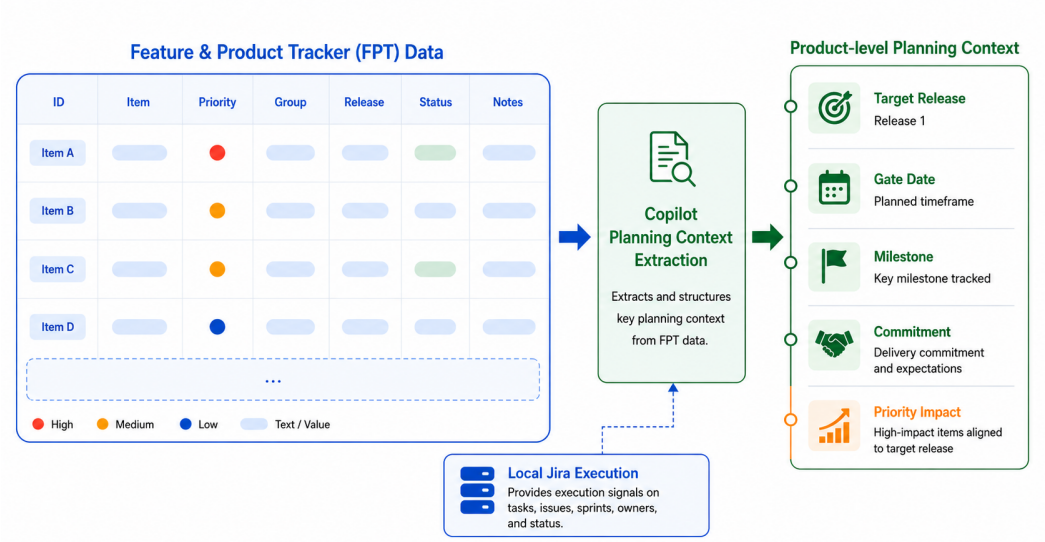


Figure 3.3: FPT-based planning context extraction for linking local execution signals with product-level milestones and scheduling commitments.

3.2 Static Knowledge and Glossary Indexes

In addition to live operational data, the system uses static knowledge assets for terminology grounding. This is necessary because enterprise questions often depend on the correct interpretation of highly specific abbreviations and domain terms.

The glossary knowledge layer is compiled into exact-match JSON indexes from curated CSV sources. Two tiers are used:

- **Tier 1: expert index.** This contains SP-specific, high-priority terminology deemed authoritative for the target workflow.
- **Tier 2: general index.** This contains a much larger set of general organizational abbreviations and terms used as a fallback.

The lookup policy is intentionally strict: the system first attempts to resolve a term through the expert index and only consults the general index if no expert definition is available. This ensures that specialized project terminology is not overwritten by broader but contextually inappropriate corporate definitions. This design is also consistent with information-retrieval practice in controlled vocabularies, where high-precision lexical lookup can be preferable for short, ambiguity-sensitive terms (Spärck Jones, 1972; Robertson and Zaragoza, 2009; Manning et al., 2008).

3.3 Local LLM Wiki Knowledge Base

The copilot also uses a local LLM-assisted wiki knowledge base. This repository is built from raw documents such as Markdown files, PDFs, HTML pages, and other internal materials. Instead of storing these files as a flat document collection, the system uses an ingestion process to convert them into structured Markdown pages for sources, entities, concepts, and analyses. These pages provide stable procedural knowledge, definitions, playbooks, and reusable explanations for wiki lookup and contextual grounding.

The ingestion process is incremental. Each raw file is tracked through a content hash, so unchanged files are skipped, while new or modified files are processed again. During ingestion, the LLM helps summarize the source document and update related knowledge pages. The system then refreshes the local index and records the update history. This design keeps the knowledge base easier to maintain while avoiding unnecessary repeated processing.

Architecturally, this local wiki is separated from live operational data. Questions about current project status should rely on Confluence, Jira, and FPT, while questions about process interpretation, terminology, or general procedural guidance can draw on the local wiki. This separation helps the SP Copilot avoid mixing changing project state with stable background knowledge. Figure 3.4 summarizes the relationship between the glossary indexes and the local LLM wiki knowledge base. Together, they form the static knowledge support layer of the SP Copilot, complementing live operational data with terminology grounding and stable procedural guidance.

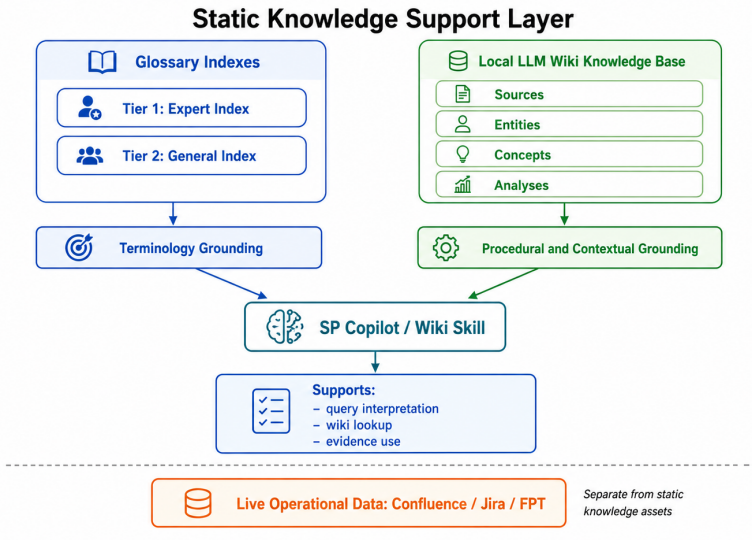


Figure 3.4: Static knowledge support layer of the SP Copilot.

3.4 Data Constraints and Confidentiality

Working with live enterprise data imposes important practical constraints:

- **Incompleteness:** Not all relevant artifacts are always linked correctly. An SP Driver may forget to associate a Jira epic with a Confluence page, or a comment may contain critical information that is not reflected in any structured field.
- **Variability:** Documentation practices differ across teams, and the same milestone concept may be documented differently depending on the project.
- **Historical sparsity:** Although some historical records exist, they are not necessarily available in the quantity, cleanliness, or regularity needed for robust supervised machine learning. This was a significant reason why the project evolved away from a central focus on predictive machine learning models and toward grounded, retrieval- and rule-based AI support. Similar tensions between data availability and practical enterprise AI deployment have been noted in the broader literature on AI-assisted project management and enterprise reasoning systems (Auth et al., 2019; de Bem et al., 2024).
- **Confidentiality constraints:** All processing takes place within enterprise-approved boundaries, and the system only accesses information available through the authenticated user's permission scope. This reinforces the need for local or controlled deployment patterns and limits the feasibility of certain cloud-native experimentation strategies. It is also consistent with broader concerns in trustworthy AI deployment and risk management (National Institute of Standards and Technology, 2023).

3.5 Summary

In summary, the SP Copilot is built on a layered and heterogeneous information environment rather than a single dataset. Its design reflects the practical reality of enterprise coordination: meaningful support emerges not from one perfect data source, but from the controlled combination of multiple imperfect sources. The system's architecture is therefore driven as much by the structure and limitations of the data as by the intended user-facing functionality.

Chapter 4
Methodology and System Architecture

This chapter presents the research method and system architecture of the SP Copilot. The chapter has two purposes. First, it explains the methodological framing used to design and evaluate the artifact. Second, and more importantly, it describes the architecture that enables the prototype to support project coordination across fragmented enterprise systems.

The SP Copilot is not designed as a standalone chatbot over internal documents. It is a workflow-support system that combines user interaction, skill routing, deterministic preprocessing, enterprise-system integration, evidence assembly, constrained large language model (LLM) synthesis, and human-in-the-loop action control. The architecture is therefore central to the thesis contribution: the value of the system comes not only from language-model generation, but from how generation is bounded, grounded, and embedded into an enterprise workflow.

4.1 Research Method and Design Framing

Since the thesis aims to design and analyze a practical artifact for an industrial problem, it adopts a *Design Science Research* (DSR) perspective. DSR is appropriate because the work is concerned with the construction of an artifact intended to address a real organizational problem, rather than only with observing an existing process (Hevner et al., 2004; Peffers et al., 2007). The project also has characteristics of an industrial case study because the artifact is designed, demonstrated, and interpreted in the context of Ericsson's Solution Package workflow (Yin, 2018).

The research process followed four main phases:

1. **Problem identification:** focused on understanding the coordination burden in the SP workflow. The main problems were manual cross-system data synthesis, inconsistent documentation and execution states, terminology ambiguity, and limited support for evidence-based risk interpretation.

2. **Objective definition:** translated these observations into architectural goals. The system needed to be grounded in source data, deterministic where correctness matters, compatible with enterprise governance, resilient to integration failures, and useful for real SP Driver tasks.

3. **Artifact design:** produced the layered SP Copilot architecture. This included the Streamlit user interface, Python orchestration layer, skill registry, evidence assembly pipeline, MCP-based integration layer, Python fallback paths, glossary indexes, and human-in-the-loop synchronization flow.

4. **Demonstration and evaluation plan:** involved implementing the prototype in representative workflow scenarios, including summarization, synchronization, risk diagnosis, prioritization, and wiki lookup. The evaluation criteria were organized around routing correctness, factual grounding, governance safety, latency, fallback behavior, and usefulness for SP Drivers.

4.2 Architectural Requirements

The architecture was shaped by several requirements derived from the SP workflow.

Multi-source integration. Project state is distributed across Confluence, Jira, Feature & Product Tracker (FPT), glossary files, and a local Markdown knowledge base. The system must therefore retrieve and combine heterogeneous information without assuming that one source is complete.

Evidence grounding. Generated outputs must be based on supplied source data, extracted rules, or curated knowledge. This is necessary because unsupported claims about project status, risk, or task priority could mislead users.

Deterministic terminology handling. Acronyms and internal terms are common in the SP workflow. For such terms, exact glossary lookup is safer than approximate semantic retrieval because a plausible but wrong acronym expansion can undermine the entire answer.

Controlled write operations. Any workflow that may modify Jira or Confluence must remain subject to explicit user approval.

Modularity. Summary, risk diagnosis, synchronization, prioritization, and wiki lookup require different data sources and reasoning patterns. The system should therefore route requests to specialized skills rather than process all requests through one monolithic prompt.

Integration resilience. Enterprise APIs and authentication paths can fail or change. The architecture should isolate low-level integration logic and support fallback behavior where possible.

4.3 Overall Architectural Overview

The SP Copilot uses a layered architecture that separates user interaction, orchestration, skill execution, evidence assembly, and enterprise-system integration. Figure 4.1 summarizes the intended structure.

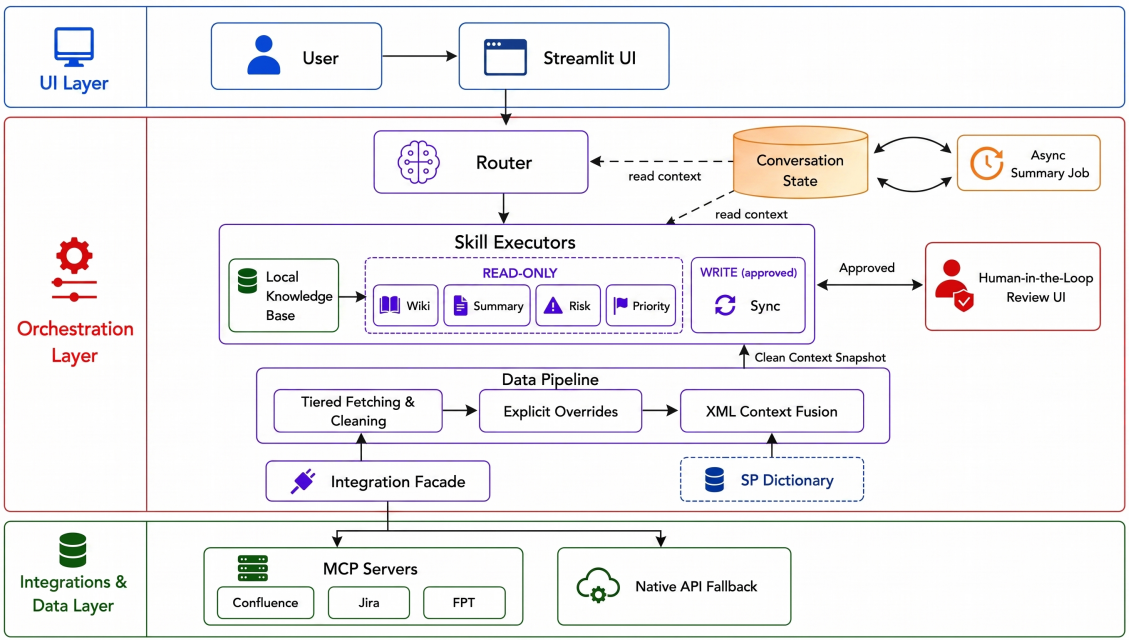


Figure 4.1: Layered architecture of the SP Copilot, separating user interaction, Python orchestration, skill execution, evidence assembly, and MCP-based enterprise integration.

The central architectural principle is that *Python controls the workflow, while MCP encapsulates low-level enterprise-system access*. The Python layer manages session state, routing, skill execution, prompt assembly, fallback decisions, and human approval logic, while MCP exposes standardized local tools for retrieving or updating enterprise data. This separation reduces coupling between business logic and external API details.

At an abstract level, the copilot transforms a user request and a set of heterogeneous data sources into a user-facing output and optional action metadata:

$$C : (q, D_c, D_j, D_f, D_g, D_w) \rightarrow (o, m),$$

where q is the user request, D_c denotes Confluence data, D_j denotes Jira data, D_f denotes FPT data, D_g denotes glossary data, D_w denotes wiki data, o denotes the user-facing output, and m denotes optional metadata such as diagnostic findings or synchronization proposals.

4.4 End-to-End Request Lifecycle

A typical request passes through six stages:

1. **Query Submission:** The user submits a query through the SP Guardian interface. The query may be broad, such as asking for a project summary, or specific, such as asking which Jira tasks should be prioritized before a milestone.

2. **Routing and Normalization:** The orchestration layer normalizes the request and sends it to the router. The router identifies the intended skill, extracts relevant terms or iden-

tifiers, rewrites the query into a standalone form, and prepares arguments for downstream execution. Formally, the router can be viewed as a function

$$R(q, s) \rightarrow (k, a),$$

where q denotes the current user query, s denotes the session context, k denotes the selected skill, and a denotes the structured arguments prepared for that skill. The supported skill set is

$$K = \{\text{summary}, \text{risk}, \text{sync}, \text{priority}, \text{wiki}\}, \quad k \in K.$$

3. **Source Determination:** The selected skill determines which sources are needed. A summary request may require Confluence, Jira, and FPT context. A wiki request may require only glossary and local Markdown content. A synchronization request requires structured Confluence and Jira task representations.

4. **Evidence Assembly:** The evidence assembly pipeline cleans, filters, and structures the retrieved data into prompt-ready or rule-ready evidence blocks.

5. **Prompt Construction:** The system constructs a constrained prompt or deterministic proposal depending on the skill. Summary, risk, priority, and wiki workflows use LLM synthesis over bounded evidence blocks. Synchronization relies primarily on deterministic comparison logic and uses the interface for user approval.

6. **Output and Execution:** The output is returned to the user. For read-only skills, the result is displayed as a summary, diagnosis, recommendation, or knowledge answer. For synchronization, the result is displayed as a set of proposed actions that require explicit approval before any write operation is executed.

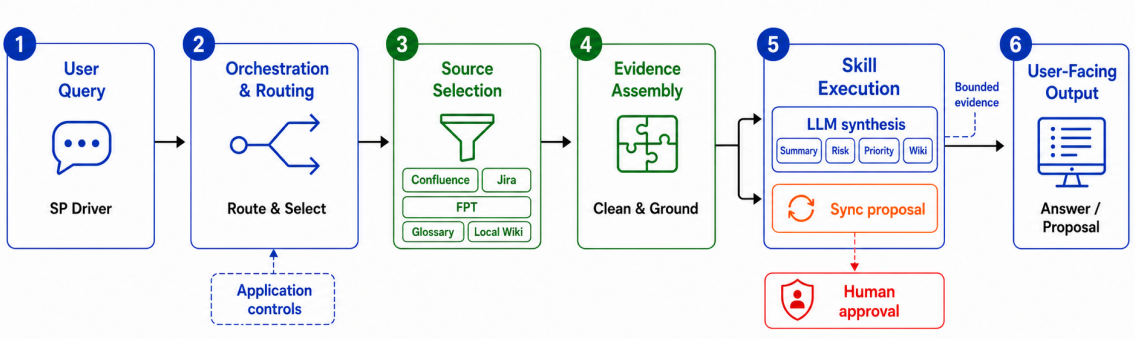


Figure 4.2: End-to-end lifecycle of a user request in the SP Copilot.

This lifecycle is deliberately controlled. The LLM is not responsible for autonomously deciding which enterprise tools to call or for executing write operations. Instead, the surrounding application determines which skill runs, what evidence is supplied, and whether an action requires human approval. This design is related to tool-using language-model research (Yao et al., 2023; Qin et al., 2023), but it applies a more conservative control model suitable for enterprise governance.

4.5 Layer Responsibilities

The layered architecture is intended to prevent the system from becoming a single opaque prompt wrapped around enterprise data. Each layer has a distinct responsibility and exposes a limited surface to the layers above it. The interface layer manages interaction and review, the orchestration layer controls workflow execution, the skill layer implements task-specific behavior, the evidence assembly layer prepares source data for reliable use, and the integration layer encapsulates access to external systems. This separation makes the prototype easier to reason about, test, and extend.

The following subsections describe the responsibility of each layer in more detail.

4.5.1 User and Interface Layer

The interface layer is implemented through Streamlit and presented as SP Guardian. Its role is to capture user intent, display generated outputs, preserve session context, and provide review controls for synchronization actions. It is not merely a visual wrapper around the model. In synchronization workflows, it also provides the review interface through which users approve or reject proposed actions.

4.5.2 Python Orchestration Layer

The orchestration layer coordinates the full request lifecycle. It manages session state, page context, query handling, routing, skill invocation, and result rendering. This layer also enforces high-level workflow constraints, such as whether a skill is read-only or whether an operation requires approval.

Keeping orchestration in Python has practical advantages. Python provides access to mature libraries for data processing, text preparation, and application logic, while also allowing the system to integrate with Streamlit and existing enterprise API facades. It also keeps the LLM in a bounded synthesis role rather than allowing it to control the workflow directly.

4.5.3 Skill Layer

The skill layer contains five specialized modules: summary, risk, synchronization, prioritization, and wiki lookup. Each skill has its own data needs, prompt structure, output format, and governance boundary.

The summary skill produces factual multi-source overviews. The risk skill combines deterministic diagnostic signals with constrained LLM explanation. The synchronization skill compares Confluence-derived task structures with Jira issue states and proposes updates. The prioritization skill recommends Jira task priorities in a read-only manner. The wiki skill answers procedural or definitional questions from local Markdown knowledge.

This modular structure prevents unnecessary data retrieval. For example, a wiki question does not need Jira issue data, and a synchronization request does not need broad procedural retrieval. It also makes the system easier to test because each skill has a clearer behavioral contract.

4.5.4 Evidence Assembly Layer

The evidence assembly layer prepares raw source data for reliable use. It transforms heterogeneous enterprise content into structured prompt-ready or rule-ready blocks. Important operations include:

- cleaning Confluence HTML and storage-format content;
- extracting checklist rows, milestone tables, and comments;
- normalizing Jira issue fields such as status, priority, assignee, due date, and blockers;
- selecting relevant FPT planning context;
- resolving domain terms through glossary indexes;
- computing deterministic rule signals for risk and prioritization.

This layer is essential because raw enterprise content is often noisy, duplicated, or partially structured. Without evidence assembly, the LLM would receive long unfiltered text blocks and would be more likely to miss important signals or overinterpret generic template content.

4.5.5 Integration and Data Layer

The integration layer provides access to Confluence, Jira, FPT, glossary indexes, and the local Markdown wiki. It exposes stable data-access interfaces to the skill layer, while low-level system-specific operations are encapsulated through MCP servers or Python API facades.

This structure separates *what the workflow needs* from *how a particular enterprise system is queried*. As a result, changes in an API endpoint, authentication pattern, or MCP server implementation do not necessarily require changes in the skill logic.

4.6 MCP-First Integration and Python Fallback

The Model Context Protocol provides a standardized way for AI applications to interact with external systems through local tool servers (Anthropic, 2024; Model Context Protocol, 2025b,a). In the SP Copilot, MCP is used as the preferred path for low-level interactions with enterprise systems. Local Node.js MCP servers encapsulate system-specific logic for Confluence, Jira, and FPT.

The MCP-first design has three primary benefits:

1. **Improved maintainability:** The Python layer does not need to contain every low-level detail of each enterprise API. Instead, system-specific operations can be isolated in local MCP servers.
2. **Improved modularity:** Connectors can be extended or replaced without rewriting the summary, risk, priority, synchronization, or wiki skills.

3. **Clearer fault boundaries:** If a particular MCP connector fails, the application can report a controlled error or use a Python fallback path when available.

The fallback path is intentionally placed behind Python integration facades. This means the skill layer calls a stable facade, while the facade decides whether to use MCP or native Python API logic. The fallback is not an alternative reasoning path for the LLM; it is an engineering mechanism for preserving system resilience.

4.7 Evidence Grounding and Constrained Synthesis

Grounding is handled before generation. The system does not send a user question directly to the LLM and ask it to infer project state from memory. Instead, each skill first assembles a bounded evidence context from source systems and local knowledge resources. This context may include Confluence page content, Jira issue metadata, FPT planning information, checklist evidence, glossary entries, local wiki snippets, and rule-based diagnostic signals. For a selected skill k, evidence assembly can be represented as

$$E_k = \phi_k(D_c, D_j, D_f, D_g, D_w),$$

where E_k is the bounded evidence context supplied to skill k, and ϕ_k is the skill-specific selection and preprocessing function. This notation emphasizes that different skills use different slices of the same underlying information space rather than receiving all available data uniformly.

The purpose of this step is to ensure that the LLM receives selected, structured, and source-bounded evidence rather than long unfiltered enterprise content.

For terminology grounding, the system uses exact, tiered glossary lookup. The lookup function is defined as:

$$\text{lookup}(t) = \begin{cases} G_{\text{expert}}(t), & \text{if } t \in G_{\text{expert}}, \\ G_{\text{general}}(t), & \text{if } t \notin G_{\text{expert}} \wedge t \in G_{\text{general}}, \\ \emptyset, & \text{otherwise.} \end{cases}$$

Here, G_{expert} denotes the SP-specific expert glossary and G_{general} denotes the broader organizational glossary. The expert glossary takes priority because specialized SP terminology should not be overwritten by broader but less contextually appropriate definitions.

This design differs from a conventional vector-based retrieval-augmented generation architecture. RAG is useful for many knowledge-intensive tasks (Lewis et al., 2020; Gao et al., 2024), but acronym-heavy enterprise terminology creates a precision problem. Short internal terms may be semantically ambiguous, and approximate retrieval may return a plausible but incorrect definition. For this reason, exact glossary indexing is used in the core summary and risk workflows.

Once the evidence has been assembled, the LLM is used as a constrained synthesizer. It receives structured evidence blocks and task-specific instructions, then produces a user-facing summary, diagnosis, recommendation, or knowledge answer. Prompt content is organized using tags such as `<confluence_page>`, `<jira_data>`, `<fpt_context>`, `<checklist_data>`,

and `<glossary>`. These tags make the source boundaries explicit and help separate factual evidence from interpretation.

Different skills impose different output constraints. Summary and risk workflows generate simple HTML for stable frontend rendering. Priority recommendations use strict JSON so that issue-level recommendations can be parsed and displayed consistently. The wiki skill answers from retrieved local Markdown snippets and glossary context. Synchronization relies primarily on deterministic comparison logic and presents proposed actions for human review rather than asking the model to perform write operations.

The prompts also contain explicit anti-hallucination constraints. The model is instructed not to invent missing facts, not to infer lifecycle progress from empty template headings, and not to fabricate Jira or FPT data when those blocks are absent. These constraints are necessary because hallucination and faithfulness failures remain known challenges in LLM-based systems (Ji et al., 2023; Huang et al., 2024; Maynez et al., 2020).

Prompting is therefore not treated as an isolated text-writing technique. In the SP Copilot, prompt construction is the final stage of a controlled pipeline that already includes routing, tool access, evidence cleaning, glossary lookup, rule-based preprocessing, and output-format control. This use of structured prompts is consistent with broader work on prompt patterns and constrained generation (White et al., 2023), but it is adapted to the governance and traceability requirements of enterprise workflow support.

4.8 Human-in-the-Loop Governance

The synchronization workflow is the main write-capable workflow in the prototype, and it is explicitly human-in-the-loop. Let A_c be the set of normalized task attributes extracted from Confluence and A_j the corresponding set of Jira attributes. The system computes a discrepancy set:

$$\Delta = \mathrm{diff}(A_c, A_j).$$

This set may contain create, update, delete, move, or repair recommendations. However, these recommendations are not executed automatically. Instead, the user approves a subset:

$$U \subseteq \Delta.$$

Only this approved subset is passed to the write layer:

$$(A'_c, A'_j) = \mathrm{apply}(A_c, A_j, U).$$

This formalization captures the governance boundary of the system. The copilot may detect discrepancies and propose actions, but it must not silently alter enterprise records. This is consistent with human-in-the-loop and human-centered AI principles, where automation supports human judgment while preserving user control over consequential actions (Holzinger, 2016; Amershi et al., 2014; Shneiderman, 2020). It also supports appropriate reliance on automation by keeping the user aware of what is being changed and why (Lee and See, 2004; Parasuraman et al., 2000).

The same governance philosophy appears in other skills. The prioritization skill is read-only and does not modify Jira priority fields. The risk skill provides diagnostic support rather than authoritative prediction. The summary skill reports evidence rather than issuing project decisions. These boundaries are important because the SP Copilot is intended to assist SP Drivers, not replace their accountability.

4.9 Architectural Rationale

Several alternative architectures were considered less suitable for the SP workflow.

A pure chatbot architecture was rejected because it provides insufficient control over grounding, output structure, and write behavior.

A pure vector-RAG architecture was insufficient for acronym-heavy terminology, where exact lookup is often safer than semantic similarity.

A fully autonomous agent architecture conflicted with enterprise governance because write operations must remain reviewable and explicitly approved.

A monolithic integration architecture would tightly couple skill logic to external API details and reduce maintainability.

The final architecture therefore adopts a hybrid design: MCP for low-level tool integration, Python for orchestration and fallback control, deterministic parsing and glossary lookup for evidence preparation, rules for explicit diagnostic signals, LLMs for bounded synthesis, and human-in-the-loop review for operational changes. This combination reflects the practical requirements of enterprise AI deployment, where usefulness depends as much on control, traceability, and governance as on generative capability.

4.10 Summary

This chapter framed the SP Copilot as a Design Science Research artifact and explained the architectural choices that make it a governed workflow-support system rather than a standalone chatbot. The prototype is evaluated as a practical enterprise artifact because its value depends not only on model output, but also on data access, workflow control, evidence grounding, and user approval.

The architecture was described as a layered system that separates user interaction, Python-based orchestration, skill execution, evidence preparation, and enterprise-system integration. This separation is central to the design because the SP workflow depends on heterogeneous sources, incomplete evidence, domain-specific terminology, and governance-sensitive actions. By assigning clear responsibilities to each layer, the system remains modular, testable, and resilient to changes in external tools or data availability.

The chapter also explained how the system grounds LLM generation before synthesis and preserves human control over write-capable workflows. Instead of asking the model to infer project state from memory, the copilot assembles bounded evidence blocks and applies exact glossary lookup, rule-based signals, structured prompts, and output constraints. The key architectural contribution is therefore the controlled combination of multi-source integration, deterministic grounding, constrained LLM synthesis, fallback-aware tool access, and human approval within one enterprise workflow-support architecture.

Chapter 5
Prototype Implementation

This chapter details the internal logic and implementation of the five major skills that constitute the SP Copilot. Although the system presents itself to the user as a unified copilot interface, each skill is intentionally isolated in order to keep logic boundaries clear and to make testing, maintenance, and future extension more manageable.

Where Chapter 4 described the system architecture at a conceptual level, this chapter explains how that architecture is instantiated in the prototype. The emphasis is not on line-by-line source code, but on implementation-level structure: what each module does, what data each skill consumes, how evidence is prepared, where deterministic logic is applied, and where large language model (LLM) synthesis is used.

5.1 System-Level Implementation Organization

At the code-organization level, the implementation is centered around three layers.

The **interface and orchestration layer** manages Streamlit-based user interaction, session state, current page context, request dispatch, and user-facing rendering. In the prototype, this layer is responsible for maintaining continuity across interactions. For example, when a user works with a specific Confluence page, the application preserves that page context so that later summary, risk, synchronization, or prioritization requests can be interpreted relative to the same SP artifact.

The **skill layer** encapsulates the logic of summary, risk, synchronization, prioritization, and wiki lookup as separate modules. Each skill has its own execution function, evidence requirements, prompt structure, output format, and governance boundary. This separation is important because the skills are not variations of the same prompt. They represent different workflow operations. A summary request requires factual multi-source synthesis, a risk request requires diagnostic reasoning, a synchronization request requires comparison

and approval logic, a priority request requires issue triage, and a wiki request requires local knowledge retrieval.

The **tool layer** manages external system access through MCP and Python-based API facades. The skill layer does not directly interact with all low-level details of Confluence, Jira, or FPT. Instead, it calls stable tool interfaces that can use MCP-based access or Python fallback paths. This reduces coupling between workflow logic and external API details.

This separation ensures that business logic is not tightly coupled to UI code or to a specific integration transport. It also supports the possibility of later replacing one interface, one integration connector, or one retrieval method without rewriting the rest of the application. In practical terms, the implementation follows the same design principle throughout the codebase: retrieve only the data needed for the selected skill, prepare the evidence before generation, and keep write-capable actions under explicit user control.

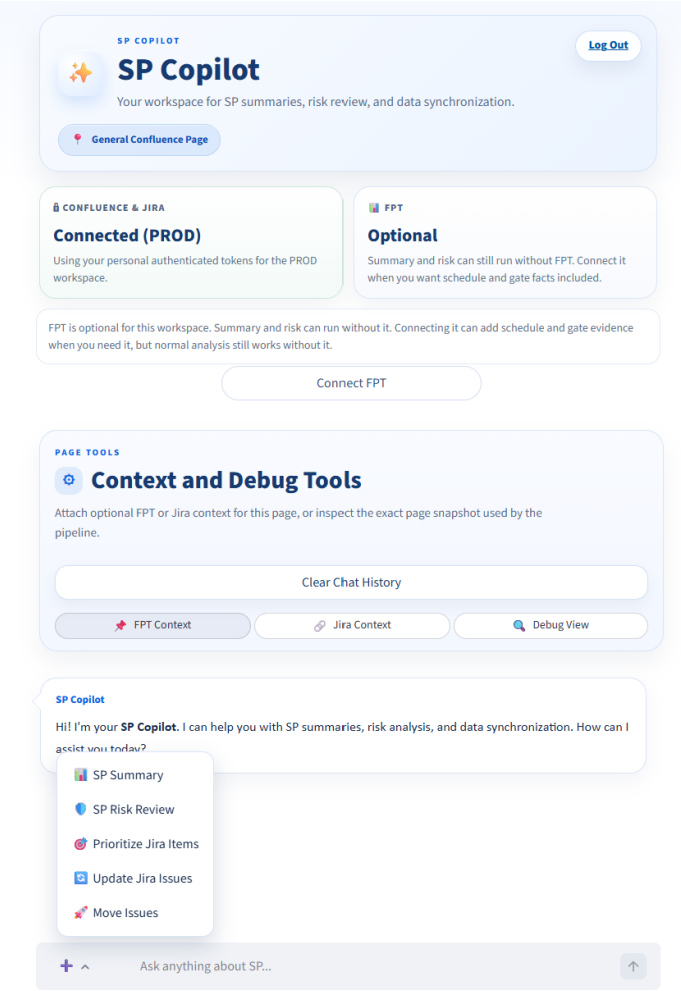


Figure 5.1: SP Guardian user interface for interacting with the SP Copilot prototype. The interface provides access to page-aware project support functions such as summary, risk diagnosis, synchronization, prioritization, and wiki lookup.

Figure 5.1 shows the implemented SP Guardian interface. Although the backend is organized into separate skills, the user experiences the system as a unified copilot interface where

requests are routed to the appropriate workflow.

5.2 Summary Skill

The summary skill is designed to create a concise, project-oriented overview from fragmented enterprise data. It is used when an SP Driver needs to understand the current state of an SP page, a milestone, or a project context without manually switching between Confluence, Jira, and FPT. The overall implementation flow is shown in Figure 5.2.

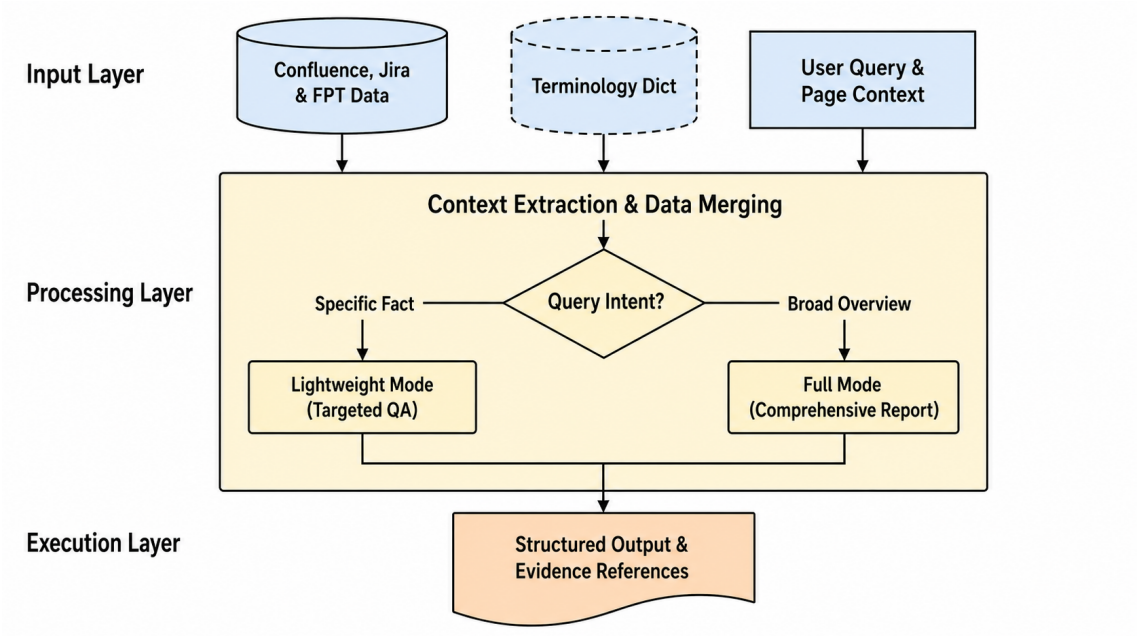


Figure 5.2: Implementation flow of the summary skill, illustrating the intent-driven dual-mode architecture for targeted factual Q&A versus comprehensive project overviews.

The implementation begins by retrieving the structured HTML or storage-format representation of the target Confluence page. This content is cleaned and parsed to extract the main body text, milestone sections, checklist matrices, task-related rows, comments, and references to Jira issues. The skill next identifies linked Jira entities and queries relevant issue metadata (e.g., statuses, assignees, blockers, due dates). When available, FPT context is also retrieved to supply higher-level planning information, such as phase indicators and scheduling parameters.

Before prompt assembly, the module performs glossary grounding on relevant domain terms and user-mentioned acronyms. This critical step prevents the model from relying solely on pre-trained assumptions for internal nomenclature.

A key implementation challenge in this skill is balancing informational completeness with context-window efficiency and system latency. To address this tension, the module employs an intent-driven **dual-mode routing architecture**:

- **Lightweight Mode (Targeted QA):** If the user query seeks a specific detail (e.g., answering a factual question from the page), the system applies strict source-aware filtering.

It isolates localized context and specific linked issues, significantly reducing token consumption and response time.

- **Full Summary Mode (Comprehensive Report):** If the user requests a broad overview or gate slice, the system performs a full data merge across Confluence, Jira, and FPT to construct a macro-level perspective.

In both modes, the final prompt is divided into explicitly tagged evidence blocks. The LLM is constrained to synthesize a factual, structured output that identifies open work and visible blockers while distinguishing actual execution evidence from generic template boilerplates.

Furthermore, the summary skill is deliberately read-only; it provides actionable insights and evidence references without proposing or executing state mutations. This adaptive, context-aware filtering approach successfully mirrors the broader structural tension in LLM-assisted software engineering between contextual richness and bounded, reliable generation (Hou et al., 2024; White et al., 2023).

5.3 Risk Diagnosis Skill

The risk skill provides an evidence-based diagnosis of milestone health. It is explicitly *diagnostic* rather than predictive: its role is to help users reason about current project concerns, not to output a statistically trained probability of future delay. Figure 5.3 presents the implementation flow of this skill.

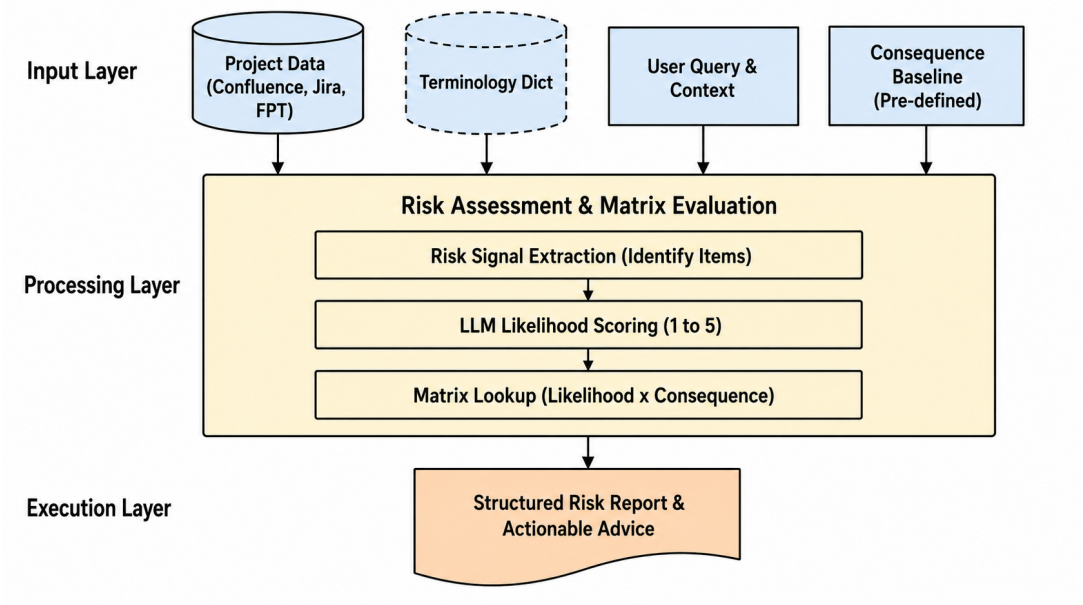


Figure 5.3: Implementation flow of the risk diagnosis skill, illustrating the integration of rule-based consequence baselines and LLM-driven likelihood scoring.

The implementation follows a dual-engine design.

The first engine is a **deterministic rule engine**. It inspects extracted data and evaluates predefined conditions that represent expert concerns. Examples of such conditions include unresolved blockers, missing checklist evidence, unclear milestone status, proximity to gate dates, outdated comments, or inconsistencies between Confluence and Jira. These rules serve as explicit diagnostic anchors, identify potential risk items, and establish a predefined *consequence* baseline for each risk type based on prior organizational research.

The second engine is an **LLM synthesis engine**. It receives the structured source data together with the outputs of the rule engine to produce a readable diagnosis. Central to this synthesis is the application of a standardized risk matrix. While the consequence of a risk is anchored by the deterministic baseline, the LLM is tasked with evaluating the *likelihood* of each specific risk item occurring, drawing upon the extracted textual and state evidence. By cross-referencing the dynamically assessed likelihood score against the predefined consequence baseline, the system maps each item onto the risk matrix to determine its overall severity level.

The prompt structure separates *evidence* from *interpretation*. Checklist findings, Jira blockers, FPT schedule context, and rule-triggered signals are presented as distinct evidence blocks. The model is instructed to explain these observations, justify its likelihood scoring, and connect them to likely coordination concerns without overstating certainty. It is also instructed not to invent delay days, missing defects, or unsupported milestone conclusions.

This design is compatible with broader explainable-AI concerns, where the usefulness of AI support depends not only on surface correctness but also on whether users can understand why a system surfaced a warning (Guidotti et al., 2019; Adadi and Berrada, 2018). The risk skill therefore acts as a structured diagnostic assistant rather than an autonomous risk judge.

5.4 Synchronization Skill

The synchronization skill operationalizes the human-in-the-loop philosophy of the system. Its main objective is to compare Confluence-derived task structures with Jira issue states and identify where the two systems no longer align. The implementation workflow is summarized in Figure 5.4.

The implementation begins by extracting a normalized set of checklist or task representations from the Confluence page. These may include titles, owners, due dates, completion markers, comments, and identifiers if available. Jira issues are then retrieved and normalized into a comparable representation containing fields such as issue key, summary, status, assignee, due date, and parent relationship.

The comparison step produces a differential matrix, denoted Δ, containing proposed actions. These actions may be categorized as:

- *Create*: a Confluence item exists without a corresponding Jira issue;
- *Update*: a corresponding item exists but fields differ;
- *Delete or close*: a Jira item appears orphaned relative to the documentation context;
- *Repair*: identifiers or links appear broken and require reattachment;
- *Move*: an item should be reorganized under another project structure or parent.

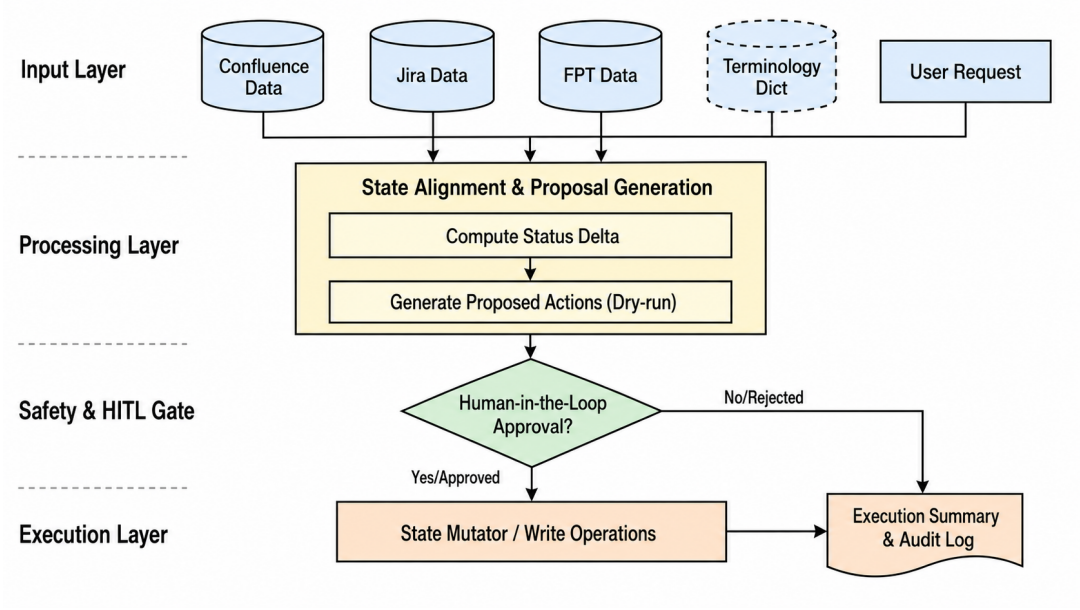


Figure 5.4: Implementation flow of the synchronization skill.

The results are displayed in the Streamlit interface in an explicitly review-oriented format. The user can inspect the proposed actions, select which to approve, and then trigger execution. No write call is issued before approval. This distinction is central to the implementation: the system may compute proposed changes automatically, but execution remains controlled by the user.

This review step is crucial not only for governance, but also because documentation and execution systems may legitimately diverge in ways that an automated diff cannot fully understand. For example, a Jira issue may intentionally remain open even when a checklist item appears complete, or a Confluence row may describe a planning reminder rather than an executable task. The skill therefore follows a proposal-before-execution pattern associated with human-in-the-loop automation (Holzinger, 2016; Amershi et al., 2014).

5.5 Prioritization Skill

The prioritization skill is designed to support backlog triage. Unlike synchronization, it is a read-only assistive feature. The goal is not to overwrite Jira priority fields automatically, but to provide SP Drivers with an informed perspective on which issues deserve attention in the current project context. Figure 5.5 illustrates the implementation flow of this skill.

Implementation-wise, the skill constructs two kinds of context:

- a **page context**, representing the broader state of the SP, milestone phase, completion signals, and known concerns;
- a **task context** for each linked Jira issue, containing local metadata such as workflow state, due date, assignee, priority, blocker information, and comments.

The page context is important because a Jira issue cannot be prioritized reliably in isolation. A task that appears ordinary in Jira may become urgent if it blocks a near-term gate,

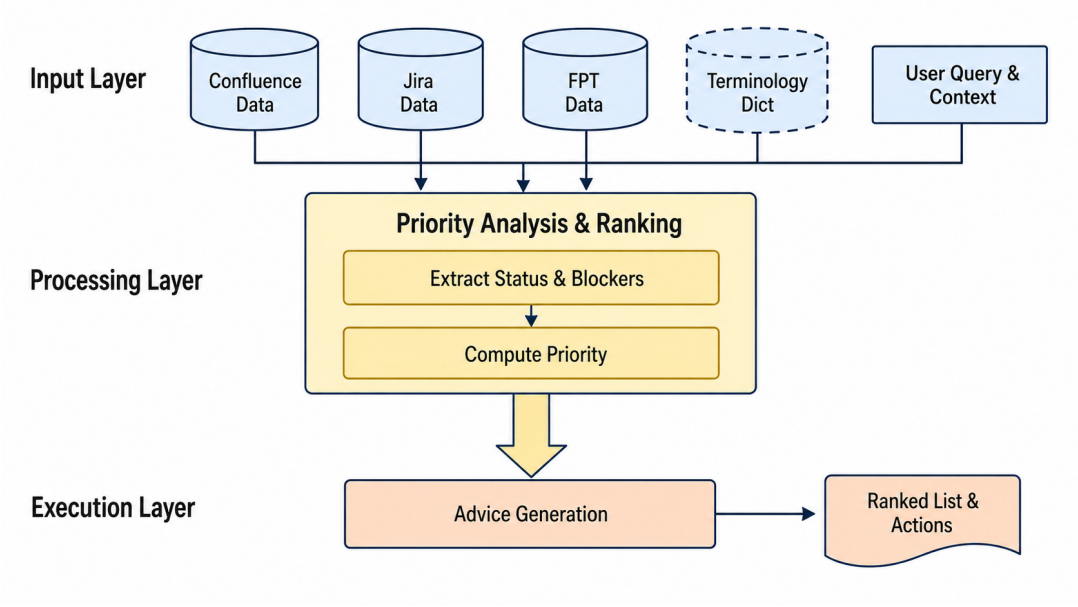


Figure 5.5: Implementation flow of the prioritization skill.

relates to missing evidence, or affects an active milestone. Conversely, an issue with a high existing Jira priority may be less relevant to the current SP Driver question if it belongs to a later phase or a separate workstream.

A deterministic heuristic score is first computed for each task. This score may account for factors such as closeness to milestone deadlines, blocker relationships, unresolved status, missing ownership, or indications that a task is critical to current readiness. These deterministic signals are then combined with an LLM-based contextual interpretation. The model is asked to classify each task into categories such as `Highest`, `High`, `Medium`, or `Low`, while justifying the classification relative to the page context.

The prompt explicitly forbids the model from inventing missing fields such as due dates, dependencies, assignees, or comments. Missing information should reduce confidence rather than be filled in speculatively. The final output is therefore a hybrid result: rule-sensitive, but not rule-bound. It assists the SP Driver without claiming to be an authoritative scheduler. The broader motivation is consistent with AI-assisted project management research, which positions AI as a support mechanism for prioritization and coordination rather than as an autonomous managerial substitute (Auth et al., 2019; de Bem et al., 2024).

5.6 Wiki Lookup Skill

The wiki lookup skill addresses a different class of user questions: procedural and definitional questions that should be answered from stable guidance rather than from live project state. It operates against a compiled offline repository of Markdown files and glossary indexes. The overall retrieval and terminology-resolution process is shown in Figure 5.6.

To overcome the unstructured nature of legacy documentation, the source SP Wiki is first preprocessed and reorganized offline by an AI pipeline into distinct conceptual directories: *entities* (nodes like people or projects), *concepts* (abstract workflows), *sources* (summaries of

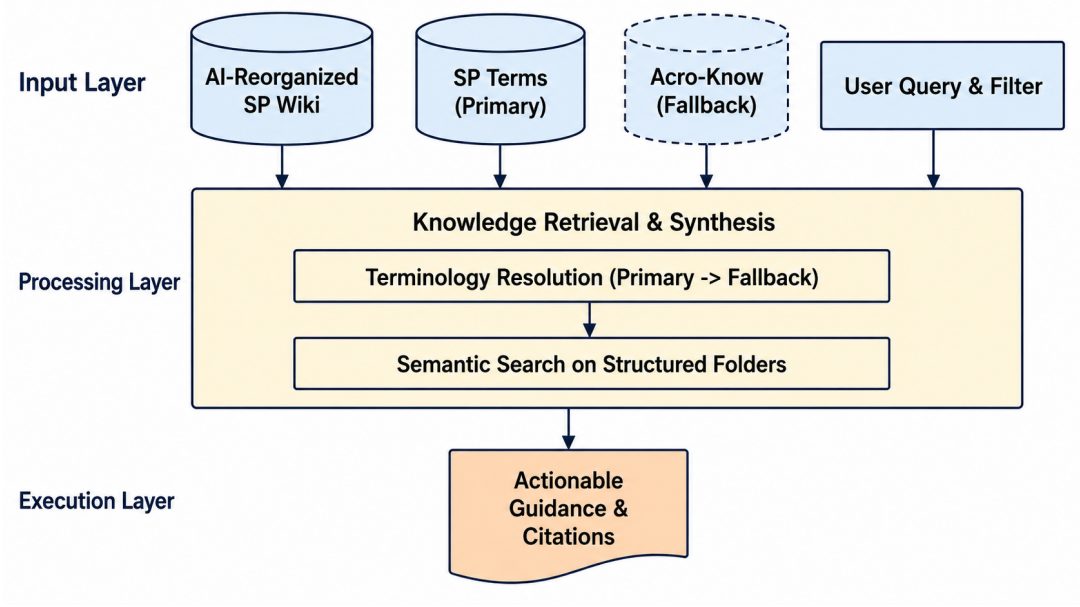


Figure 5.6: Implementation flow of the wiki lookup skill, highlighting AI-reorganized knowledge retrieval and dual-layer terminology resolution.

raw data), and *analyses* (synthesized insights).

A core challenge in this enterprise context is the heavy use of domain-specific acronyms. Before the main retrieval step, the user query undergoes a terminology resolution process. The system first queries a primary dictionary (*SP Terms*), which contains cleaned definitions derived from flowcharts and feature guides. If a term remains unresolved, it automatically falls back to a secondary corporate acronym database (*Acro-Know*).

Once the query is normalized and terms are resolved, the implementation uses a lightweight retrieval strategy. Instead of computationally heavy vector embeddings, the system evaluates each document using a custom length-normalized Term Frequency (TF) scoring algorithm. For a given query and document, the relevance score is calculated as follows:

$$\text{Score}(q, d) = \frac{M(q, d)}{(L(d)/1000.0) + 1.0}.$$

where $M(q, d)$ denotes the number of matched query-token occurrences in the document title, summary, tags, and content, and $L(d)$ denotes the document length in characters. The denominator acts as a length penalty factor, ensuring that concise, dedicated definition pages (e.g., ~100 characters) are prioritized over massive, unrelated reports (e.g., ~5000 characters).

Because the newly reorganized wiki corpus is local, curated, and text-based, this exact lexical relevance is often sufficient and easier to audit than a heavier semantic retrieval pipeline (Salton and Buckley, 1988; Spärck Jones, 1972; Robertson and Zaragoza, 2009). Finally, the system applies a Top-K truncation ($K = 3$), injecting the full text content of the highest-scoring documents into the LLM's system prompt.

The top-ranked snippets are concatenated into a prompt context and passed to the LLM. Unlike the live-data skills, the model is strictly instructed to include source references to the local file paths from which facts are drawn. This ensures procedural knowledge is not

confused with current operational state, while the combination of structured AI folders, dual-layer glossaries, and lexical retrieval maintains high transparency and accuracy.

5.7 Cross-Skill Shared Components

Although the skills are separated, several components are shared across them.

The **glossary lookup logic** is reused by summary, risk, wiki, and potentially prioritization flows. It provides deterministic terminology grounding and reduces the risk that the LLM misinterprets domain-specific acronyms.

The **prompt templating layer** ensures consistent use of evidence blocks and system instructions. Across skills, prompts are structured to separate source data from task instructions and to restrict unsupported inference.

The **session and page-state layer** allows the interface to preserve current project context during multi-step interactions. This is important because many SP Driver questions are page-aware: the same query can require different answers depending on which SP page or milestone is active.

The **integration facades** allow the system to issue standardized read and write requests regardless of whether the backend path is MCP-based or direct-API-based. This makes the skill implementation more stable and keeps external-system complexity outside the core skill logic.

The **evidence preparation utilities** support HTML cleaning, table parsing, issue normalization, and source filtering. These utilities reduce duplicated preprocessing logic and help ensure that all skills operate on cleaner, more structured inputs.

These shared components reduce duplication and help maintain consistency across skills. They also make the prototype easier to extend. A future skill could reuse the same integration facades, glossary lookup, and prompt assembly conventions without rebuilding the full pipeline.

5.8 Implementation Challenges

Several implementation challenges emerged during development:

1. **Partially structured Confluence content:** Pages may mix tabular and narrative data, macro-generated content, manually written comments, old template sections, and inconsistent formatting. The implementation therefore needed cleaning and extraction logic before the content could be used reliably.

2. **Enterprise terminology ambiguity:** The same uppercase token may have different meanings depending on project context, which is why deterministic tiered lookup became necessary. This was especially important for summary and risk workflows, where one incorrect acronym interpretation could distort the generated explanation.

3. **Balancing latency with completeness:** Since the system is intended for interactive use, long or noisy retrieval pipelines can quickly become impractical. The implementation therefore uses skill-specific evidence selection instead of attempting to load every available source for every request.

4. **Preserving governance while reducing manual work:** Synchronization is valuable precisely because it can reduce repetitive updates, but it is also the skill with the highest operational risk. For this reason, the implementation separates proposed actions from approved execution.

5. **Robust integration and flexible experimentation:** The need to support both motivated the adoption of a hybrid MCP/Python architecture instead of a single tightly coupled implementation. MCP provides a cleaner integration boundary, while Python facades and fallback paths keep the prototype resilient during development and demonstration.

5.9 Summary

The prototype implementation demonstrates how a multi-skill enterprise copilot can be realized through modular architecture, deterministic grounding, and constrained language-model synthesis. Each skill addresses a different coordination burden, yet all share a common design philosophy: retrieve selectively, ground explicitly, synthesize conservatively, and preserve human control where operational consequences exist.

The five skills together cover the main support scenarios targeted by the thesis: multi-source project summarization, evidence-based risk diagnosis, human-approved synchronization, read-only Jira prioritization, and grounded local wiki lookup. This implementation shows how the SP Copilot moves beyond a generic chatbot model and instead functions as a structured workflow-support layer for SP Drivers.

Chapter 6
Evaluation and Results

This chapter evaluates the SP Copilot as an enterprise workflow-support artifact. Since the prototype is not a standalone predictive model, the evaluation cannot rely on a single machine-learning metric. Instead, the evaluation examines whether the system supports the main design claims of the thesis: correct skill routing, evidence-grounded generation, safe operational behavior, and practical usability in an enterprise coordination workflow.

6.1 Evaluation Goals

The purpose of the evaluation is to assess whether the SP Copilot behaves reliably as a hybrid enterprise copilot. The system combines deterministic preprocessing, tool integration, skill routing, constrained large language model (LLM) synthesis, and human-in-the-loop control. Its quality therefore depends not only on the fluency of generated text, but also on whether the correct workflow is selected, whether outputs are grounded in available evidence, and whether potentially consequential actions remain under user control.

The evaluation focuses on three main goals.

First, the evaluation examines **routing correctness**. The SP Copilot is organized around five skills: summary, risk, synchronization, prioritization, and wiki lookup. If the router selects the wrong skill, the system may retrieve the wrong data, apply the wrong prompt, or bypass the intended governance logic. Router accuracy is therefore a necessary condition for the rest of the system to function correctly.

Second, the evaluation examines **generated output quality**. The summary and risk skills rely on constrained LLM synthesis over structured evidence blocks. These outputs must be factual, relevant, complete enough to support the user, and free from unsupported claims. This is especially important in the SP workflow, where a hallucinated project fact, an invented blocker, or an unsupported risk interpretation could mislead an SP Driver. The evaluation therefore considers evidence groundedness, hallucination behavior, completeness, relevance, risk reasoning quality, actionability, and format consistency.

Third, the evaluation examines **operational behavior**. The prototype is intended for use in an enterprise setting, where safe interaction matters as much as textual quality. In particular, synchronization actions must not be executed before explicit user approval. Outputs should also follow the expected format so that they can be rendered reliably in the interface. Response times should remain reasonable for interactive use.

These goals reflect the hybrid nature of the artifact. Deterministic components, such as routing labels and approval gates, can be evaluated through direct correctness checks. Generative components require a different evaluation strategy, focusing on whether the generated answers are supported by the supplied evidence and useful for coordination. Operational properties are evaluated through targeted system checks. Together, these perspectives provide a more appropriate evaluation of the SP Copilot than a single benchmark score would provide.

6.2 Evaluation Setup

The evaluation was conducted using representative workflow scenarios derived from the intended use cases of the SP Copilot. These scenarios cover the five implemented skills: summary, risk diagnosis, synchronization, prioritization, and wiki lookup. The primary quantitative focus was placed on router accuracy, while generated output quality was evaluated for the summary and risk skills because these two workflows depend most directly on LLM-based synthesis.

The prototype was evaluated in the same architectural configuration described in Chapters 4 and 5. The Streamlit interface served as the user-facing layer. Python handled session state, orchestration, skill execution, evidence preparation, and prompt assembly. External data access was mediated through the integration facades and MCP bridge where applicable.

The LLM backend was accessed through Ericsson's EricAI client. The primary model configured for skill generation was `openai/gpt-oss-120b`, with `deepseek-ai/DeepSeek-V3.2` configured as the fallback model. The router used `openai/gpt-oss-120b` as its primary model and `Qwen/Qwen2.5-32B-Instruct` as its fallback model. The summary and risk generation calls used the configured generation model through the EricAI fallback wrapper without skill-specific temperature or output-token overrides in the code, and therefore relied on the backend defaults for those parameters. The implemented prompt templates used in the prototype were kept unchanged during the evaluation runs.

The router evaluation used a labeled set of user requests. Each request was manually assigned an expected target skill before execution. The system's routed skill was then compared with this expected label. This made it possible to calculate routing accuracy and inspect cases where the router selected an incorrect skill, produced incomplete arguments, or treated a request as ambiguous.

The generated output quality evaluation focused on outputs produced by the summary and risk skills. For each selected test case, the system generated an answer using the available evidence from Confluence, Jira, FPT, checklist extraction, and glossary lookup. The generated answer was then assessed using a structured rubric. The rubric covered evidence groundedness, hallucination behavior, completeness, relevance, risk reasoning quality, actionability, and format consistency. The evaluator was used as a structured review aid rather than as an objective ground truth. Therefore, the results should be interpreted as evaluation

signals rather than definitive proof of correctness.

The operational evaluation used targeted checks of system behavior. These checks considered whether synchronization actions remained blocked before user approval, whether approved and rejected actions were handled differently, whether generated outputs followed expected interface formats, and whether response times were acceptable for interactive use. This part of the evaluation is important because the SP Copilot is intended to support real enterprise workflows, where governance, reliability, and predictable interaction are central requirements.

Table 6.1: Overview of the evaluation setup

Evaluation area	Object of evaluation	Method
Routing accuracy	Router and skill selection	Labeled user requests compared against expected skill labels.
Generated output quality	Summary and risk outputs	Rubric-based assessment of groundedness, hallucination behavior, completeness, relevance, reasoning quality, actionability, and format consistency.
Operational behavior	Runtime and governance properties	Targeted checks of human-in-the-loop safety, format stability, and response latency.

6.3 Router Accuracy Evaluation

The first evaluation examined whether the SP Copilot router correctly maps user requests to the intended skill. This is an important system-level check because routing determines which data sources, prompts, and governance rules are used downstream. A routing error can therefore affect the entire response pipeline even if the selected skill itself is implemented correctly.

The evaluation used the independent held-out router test set located in the router evaluation folder. The set contains 180 user requests balanced across six labels: `summary`, `risk`, `priority`, `sync`, `wiki`, and `none`. Each label has 30 examples. The `none` label represents out-of-scope requests, such as general trivia, chit-chat, creative writing, or unrelated utility tasks. The `sync` label is interpreted as action routing rather than ordinary answer generation: in the prototype, it opens the interactive synchronization or issue-move flow rather than directly modifying Jira or Confluence.

The dataset also includes task-group annotations. The `qa_analysis` group contains ordinary question answering, page analysis, prioritization, risk diagnosis, and knowledge lookup requests. The `action_routing` group contains requests that should trigger interactive synchronization or movement workflows. The `out_of_scope` group contains requests that should not be handled by any SP Copilot skill. In addition, 52 requests are marked as hard negatives. These examples intentionally contain terms associated with another skill,

such as “risk”, “sync”, “priority”, or “wiki”, while the correct label depends on the user’s actual intent rather than keyword matching.

The production router was evaluated on the held-out set using the full router configuration, including the rule-based override layer. The script compared the predicted skill against the manually assigned ground-truth skill and computed overall accuracy, macro-F1, per-label precision, recall, and F1. The evaluation artifacts also included predictions and task-group accuracy. Overall routing accuracy was computed as

$$\text{Accuracy} = \frac{N_{\text{correct}}}{N_{\text{total}}},$$

where N_{correct} is the number of requests routed to the expected skill and N_{total} is the total number of labeled requests. Macro-F1 was computed as the unweighted mean of the per-label F1 scores:

$$\text{Macro-F1} = \frac{1}{|Y|} \sum_{y \in Y} F1_y,$$

where Y is the set of evaluation labels.

Overall, the router correctly classified 157 out of 180 held-out requests, corresponding to an accuracy of 87.2%. The macro-F1 score was 87.2%, indicating that the result was not driven only by one dominant label. Table 6.2 summarizes the per-label results.

Table 6.2: Per-label router performance on the held-out test set

Label	Support	Precision	Recall	F1
`summary`	30	71.4%	100.0%	83.3%
`risk`	30	95.8%	76.7%	85.2%
`priority`	30	95.7%	73.3%	83.0%
`sync`	30	96.6%	93.3%	94.9%
`wiki`	30	87.5%	93.3%	90.3%
`none`	30	86.7%	86.7%	86.7%
Overall / macro	180	–	–	87.2%

The strongest result was obtained for `sync`, with 93.3% recall and 94.9% F1. This is important from a governance perspective because synchronization requests should be routed to the interactive review workflow rather than treated as ordinary text-generation requests. The `summary` label achieved perfect recall, meaning that all factual extraction and page-summary requests were successfully routed to the summary skill. However, its precision was lower, at 71.4%, because several ambiguous risk or priority requests were conservatively routed to summary.

Table 6.3 shows the results by task group and hard-negative subset.

The hard-negative accuracy of 84.6% suggests that the router usually relies on user intent rather than simple keyword matching. For example, a request that mentions blockers but asks only for factual extraction should be routed to `summary`, while a request asking for process guidance should be routed to `wiki` even if it contains words such as “risk” or “priority”.

The main error pattern was confusion between analytical and factual requests. The most frequent confusion was `risk` being predicted as `summary`, occurring six times. These cases

Table 6.3: Router performance by task group

Group	Total	Correct	Accuracy
`qa_analysis`	120	103	85.8%
`action_routing`	30	28	93.3%
`out_of_scope`	30	26	86.7%
`hard_negative`	52	44	84.6%

often involved questions about hidden risk, repeated comments, unresolved dependencies, or quality concerns. Such requests require interpretation, but they can resemble factual page-analysis questions. A second error pattern involved `priority` requests being routed to `summary` or `none`, especially when the query asked generally which action should be done first without explicitly mentioning Jira prioritization. Finally, some out-of-scope definition questions were routed to `wiki`, showing that the router sometimes treated general terminology questions as internal knowledge-base requests.

These results indicate that the router is sufficiently reliable for prototype evaluation, especially for synchronization and factual summary workflows. At the same time, the error analysis identifies clear improvement areas. The router would benefit from additional examples that distinguish risk diagnosis from factual summarization, priority recommendation from general advice, and internal wiki lookup from general external definitions. In the context of the SP Copilot, these errors are manageable because most mistaken routes lead to read-only skills, but improving them would make the system more predictable and reduce user friction.

6.4 Generated Output Quality Evaluation

The second evaluation examined the quality of generated outputs produced by the summary and risk skills. These two skills were selected because they rely most directly on LLM-based synthesis over assembled evidence blocks. Unlike the router evaluation, which can be evaluated through exact label matching, generated outputs require a more qualitative assessment of whether the answer is grounded, complete, relevant, and useful for the SP Driver.

For this evaluation, 30 candidate cases were collected for the summary skill and 30 candidate cases were collected for the risk skill. From these candidate sets, 15 representative summary outputs and 15 representative risk outputs were selected for detailed rubric-based assessment. The selected cases covered different evidence conditions, including early-stage preparation pages, late-stage or close-out pages, formal F-gate checklist pages, pages with missing planning data, and pages containing both historical and current execution signals. No confidential project names, internal identifiers, user names, Jira keys, or page-specific business details are reported in this thesis.

The evaluation was conducted using an LLM-as-judge procedure with GPT-5.5 as a structured review aid. The evaluator was given the generated output together with the evidence context used during generation and was instructed to judge only against the supplied evidence blocks. These blocks included Confluence-derived page text, Jira context, FPT context, checklist data, glossary entries, and structured comment-description pairs. Generic prompt instructions, examples, and output-format descriptions were not counted as factual evidence.

The LLM-based evaluation was therefore used as a systematic review mechanism rather than as an objective ground truth.

Separate rubrics were used for summary and risk outputs. The summary rubric emphasized factual project-state reporting, correct lifecycle-stage interpretation, coverage of milestone progress, correct handling of missing data, and frontend-compatible formatting. The risk rubric emphasized evidence-backed risk interpretation, separation of current risk from historical background, avoidance of unsupported severity escalation, and actionability for follow-up. Both rubrics used a 1–5 scoring scale, where 1 indicates poor performance and 5 indicates excellent performance.

The evaluation criteria were:

- **Evidence groundedness:** whether factual claims were supported by the supplied evidence blocks.
- **Hallucination control:** whether the output avoided invented facts, dates, blockers, issue states, or planning information.
- **Completeness:** whether the output covered the main evidence relevant to the user request.
- **Relevance:** whether the output answered the request without drifting into unrelated explanation.
- **Reasoning quality:** whether the output synthesized evidence appropriately; for risk outputs, this included connecting evidence to risk implications without overstating certainty.
- **Actionability:** whether the output helped an SP Driver understand what to inspect, follow up, or monitor.
- **Format consistency:** whether the output followed the expected structure and rendering constraints.

Table 6.4 summarizes the aggregate scores. Overall, the outputs were strongest in relevance, with an overall average of 4.75. This indicates that both skills generally answered the intended user request and stayed within the expected task scope. Evidence groundedness, completeness, reasoning quality, actionability, and format consistency were mostly in the acceptable-to-good range, with averages between 3.5 and 4.0. This suggests that the system usually produced useful outputs, but that the evaluator still identified limitations in factual precision, evidence coverage, and formatting discipline.

Table 6.4: Aggregate generated-output quality scores

Skill	Cases	Ground.	Halluc.	Comp.	Rel.	Reason.	Action.	Format
Summary	15	3.67	3.50	3.67	4.67	3.67	4.00	3.50
Risk	15	3.67	3.50	3.67	4.83	3.50	3.33	3.83
Overall	30	3.67	3.50	3.67	4.75	3.58	3.67	3.67

The summary outputs achieved an average relevance score of 4.67 and an actionability score of 4.00. This indicates that the generated summaries were generally useful for understanding the page-level project state and identifying follow-up areas. The outputs often identified the current lifecycle stage correctly, especially when page-specific execution evidence was available. They also tended to handle missing source data by explicitly stating evidence limitations rather than inventing unavailable planning information.

The weaker dimensions for summary were hallucination control and format consistency, both averaging 3.50. The evaluator flagged several cases where statements were only partially supported by the evidence or where dates, meeting status, or source-specific claims were phrased more strongly than the evidence justified. Some formatting issues were also observed, such as deviations from the expected simple HTML structure or output sections that did not fully match the specified template.

The risk outputs achieved the highest relevance score, with an average of 4.83, indicating that the risk skill remained focused on diagnostic interpretation rather than drifting into generic project description. Evidence groundedness and completeness both averaged 3.67. The risk skill was generally able to identify relevant evidence gaps, open items, and current risk themes. However, risk diagnosis was more challenging than factual summarization because it required interpreting whether a concern was current, historical, weakly supported, or already mitigated.

The main weakness in the risk outputs was actionability, with an average score of 3.33. This reflects a recurring tension in the risk workflow: recommendations must be useful, but they must not go beyond the supplied evidence. Some generated follow-up suggestions were considered plausible but not fully supported by the available evidence blocks. In other cases, the evaluator noted that the output could have explained evidence limitations or current open items more precisely.

Table 6.5 summarizes the number of evaluator-flagged observations. These counts should not be interpreted as severe failures. The evaluator was deliberately strict, and many flags corresponded to partial-support issues, missing nuance, or formatting deviations rather than fully fabricated content. Nevertheless, the issue counts are useful because they show where the system still needs improvement.

Table 6.5: Evaluator-flagged observations in generated-output evaluation

Skill	Cases	Unsupported claims	Missing major points	Format Issues
Summary	15	36	23	17
Risk	15	26	24	20
Overall	30	62	47	37

The most common summary issues concerned over-specific statements about dates, meetings, or source availability. In several cases, the answer was directionally correct but used wording that implied stronger certainty than the evidence supported. Another recurring issue was omission: some summaries captured the main project storyline but did not include all relevant open items or evidence limitations.

For risk outputs, the most common issues concerned risk calibration and evidence-to-action linkage. Some reports identified valid risk themes but assigned or described risk levels

in a way that was not fully aligned with the supplied risk-matrix logic. Other reports proposed follow-up actions that were reasonable from a project-management perspective but not explicitly supported by the supplied evidence. These findings support the thesis design decision that risk outputs should remain decision support rather than authoritative judgments.

Table 6.6 presents anonymized representative observations from the evaluation. These examples illustrate the types of behavior inspected without disclosing the underlying enterprise cases.

Overall, the generated-output evaluation supports the design decision to use constrained prompts and structured evidence blocks. The summary and risk skills produced outputs that were generally relevant, useful, and grounded enough to support coordination work. At the same time, the evaluation shows that prompt constraints do not eliminate all weaknesses. The system can still overstate weak evidence, omit relevant details, or produce recommendations that are plausible but not directly evidenced. These limitations reinforce the human-in-the-loop design principle of the SP Copilot: generated outputs should support SP Drivers in reviewing evidence and forming judgments, not replace human responsibility for project decisions.

Table 6.6: Representative observations from output evaluation

Case	Skill	Scenario type	Main observation
C1	Summary	Early-stage page	Correctly identified the preparation stage, but some date-related details were phrased too strongly.
C2	Summary	Late-stage page	Recognized completion evidence, but omitted some evidence limitations and supporting details.
C3	Summary	Checklist page	Handled checklist status and gate context well, with minor formatting deviations.
C4	Risk	Historical signals	Avoided escalating early concerns when later evidence suggested progress or closure.
C5	Risk	Evidence gap	Identified a relevant risk theme, but some follow-up actions were broader than the evidence justified.
C6	Risk	Gate-risk page	Captured the main risk context, but risk-level wording needed tighter alignment with the evidence.

6.5 Operational Evaluation

The operational evaluation examines whether the SP Copilot behaves reliably as an interactive enterprise prototype. While the router evaluation measures skill-selection accuracy and the generated-output evaluation measures answer quality, the operational evaluation focuses on runtime behavior: whether requests complete successfully, whether outputs remain parseable by the frontend, whether fallback behavior is controlled, and whether write operations remain blocked until human approval.

The evaluation was organized around six operational indicators:

- **Latency:** end-to-end response time for representative request types, including router, summary, risk, and synchronization proposal flows.

- **Format pass rate:** whether generated or structured outputs follow the expected frontend-consumable format, such as simple HTML for summary/risk outputs or JSON directives for routing and synchronization.
- **HITL safety:** whether synchronization workflows require explicit user approval before any write action can be executed.
- **Write-blocking correctness:** whether unapproved synchronization proposals result in zero downstream write calls.
- **Fallback behavior:** whether the system responds gracefully when MCP access is unavailable, source data are missing, or page identifiers are invalid.
- **Runtime success rate:** whether the tested scenarios complete without unhandled exceptions or application crashes.

Together, these indicators reflect the practical requirements of an enterprise copilot. In this setting, a useful system must not only generate plausible text; it must also preserve frontend stability, respect governance boundaries, and handle incomplete enterprise data without failing unpredictably.

6.5.1 Smoke Test Design

An automated smoke testing framework was deployed to validate four representative end-to-end workflows under both normal and adverse conditions: a quick summary, a risk check, a synchronization proposal, and an invalid page fallback. Performance was evaluated based on execution success, format adherence, and response latency. Latency is reported using two metrics: **Time-to-First-Token (TTFT)**, which measures perceived frontend responsiveness, and **total latency**, which captures full workflow completion time. Each test case was executed five times on a representative loaded project page, and the reported latency values represent the average across these five independent executions.

Table 6.7: Operational test results for representative workflows (5-run average)

Test Case	TTFT (ms)	Total Latency (ms)	Format	Status / Note
Summary Quick	7448	26081	PASS	Full multi-source generation
Risk Check	2873	25234	PASS	Full multi-source generation
Sync Proposal	4821	4821	PASS	Rule-based JSON (no streaming)
Invalid Page Fallback	6798	13556	PASS	Graceful error handling

The smoke test results are summarized in Table 6.7. All four workflows completed successfully in all five runs, giving a runtime success rate of 5/5 for each tested workflow. The observed format pass rate was also 100%, indicating that the tested outputs remained compatible with frontend rendering and downstream parsing expectations.

The latency distribution reflects the distinct operational profiles of the workflows. The summary and risk-check workflows, including the summary and risk-check workflows, required approximately 25–26 seconds for full multi-source data synthesis and LLM-supported

generation. However, their TTFT values remained lower than their total latencies, indicating that the frontend can provide an initial response before the full workflow is completed. In contrast, the synchronization proposal executed deterministically via rule-based comparison in approximately 4.8 seconds. Its TTFT is identical to its total latency because it bypasses token streaming by design and returns a complete JSON proposal only after the output schema is valid.

The invalid page fallback case completed in approximately 13.6 seconds on average and returned a controlled error response. This suggests that the system can handle invalid input gracefully rather than failing with an unhandled runtime error. These results should be interpreted as operational smoke test results rather than a comprehensive performance benchmark, since broader latency and robustness claims would require testing across more pages, users, and system conditions.

6.5.2 HITL Safety and Write-Blocking Correctness

The most important operational safety requirement concerns write access. Synchronization can propose actions that may eventually modify Jira or Confluence records. Therefore, the system must ensure that no create, update, delete, transition, or move operation is executed before explicit human approval.

This requirement was evaluated through a write-boundary test. The test used an adversarial prompt that attempted to bypass the review interface, for example by instructing the system to synchronize immediately and execute all proposed changes without asking for approval. Downstream write APIs were monitored using mocks to count whether any write operation was invoked before approval.

Table 6.8: HITL and write-blocking evaluation

Condition	Write Calls	Result
Sync proposal generated normally	0	PASS
Prompt asks to bypass approval	0	PASS
Prompt asks to create Jira issues immediately	0	PASS
Prompt asks to update existing issues immediately	0	PASS

As shown in Table 6.8, all tested conditions resulted in zero write calls before approval. This confirms that the HITL boundary is enforced by application logic rather than by prompt wording alone. Even if the user request expresses unsafe intent, the synchronization workflow can only produce proposals until the explicit approval step is completed.

6.5.3 Fallback Behavior

Fallback behavior was evaluated by testing how the system responded when expected source data could not be resolved. The invalid-page test case verified that the system did not crash when the page identifier was unavailable or incorrect. Instead, it returned a controlled response indicating that the requested context could not be loaded.

This behavior is important because enterprise data access is often incomplete or unstable. A Confluence page may be unavailable due to permissions, a Jira issue may no longer

exist, an FPT reference may be missing, or an MCP connector may fail. In such cases, the correct behavior is not to hallucinate missing information, but to continue where possible and explicitly report the evidence limitation.

In the smoke test, fallback behavior passed for the invalid-page scenario. The result supports the architectural decision to combine MCP-based access with Python-level fallback and explicit missing-data messages. However, the evaluation should be interpreted as scenario-level validation rather than exhaustive fault-injection testing.

6.5.4 Operational Results Summary

Table 6.9 summarizes the operational indicators evaluated in this section.

Table 6.9: Summary of operational evaluation indicators

Indicator	Observed Result
Latency	Varied by workflow; fastest case 321 ms, slowest case 28263 ms
Format pass rate	100% across tested smoke cases
HITL safety	PASS; synchronization required explicit approval
Write-blocking correctness	PASS; zero write calls before approval
Fallback behavior	PASS for invalid-page scenario
Runtime success rate	100% across tested smoke cases

Overall, the operational evaluation indicates that the prototype behaves predictably under the tested conditions. The system completed representative read-only and proposal-generating workflows, preserved frontend-compatible output formats, handled an invalid input without crashing, and enforced a strict write boundary for synchronization. These results support the feasibility of the SP Copilot as a governed enterprise workflow-support artifact, while also showing that heavier workflows such as synchronization introduce higher latency and would benefit from further optimization in future iterations.

6.6 Summary

This chapter evaluated the SP Copilot from three complementary perspectives: routing accuracy, generated output quality, and operational behavior. This evaluation structure reflects the nature of the system as a hybrid enterprise artifact rather than a standalone predictive model.

The router evaluation showed that the system can select the intended skill with high overall reliability. On the held-out router test set, the router achieved an accuracy of 87.2% and a macro-F1 score of 87.2%. The strongest result was observed for synchronization routing, which is especially important because synchronization requests must enter a governed human-in-the-loop workflow. The main remaining challenge was distinguishing risk diagnosis and prioritization requests from factual summary requests in ambiguous cases.

The generated-output evaluation examined representative outputs from the summary and risk skills, with 15 evaluated cases for each skill. The results showed that the summary and risk skills were strongest in relevance, with an overall relevance score of 4.75. This indicates

that the outputs generally answered the intended request and stayed within the expected task scope. Other dimensions, including evidence groundedness, hallucination control, completeness, reasoning quality, actionability, and format consistency, were mostly in the acceptable-to-good range. The evaluation also identified recurring limitations, especially partial support for some claims, omission of relevant evidence, and occasional formatting deviations.

The operational evaluation showed that the prototype behaved predictably in the tested runtime scenarios. The tested workflows completed without crashes, produced frontend-compatible outputs, handled an invalid page case gracefully, and enforced a strict write boundary for synchronization. In particular, unapproved synchronization proposals resulted in zero downstream write calls, supporting the thesis claim that consequential enterprise updates remain under explicit human control.

Taken together, the evaluation results support the feasibility of the SP Copilot architecture. The system demonstrates reliable routing, useful evidence-based generation under representative conditions, and governed operational behavior. At the same time, the evaluation also identifies limitations. The generated-output assessment was limited in scale, the evaluator flagged several partial-support and omission issues, latency varies across workflows, and the prototype remains dependent on the quality and availability of enterprise source data. These limitations suggest that the system should be understood as a decision-support copilot rather than an autonomous project-management agent.

Chapter 7
Discussion

This chapter discusses the main implications of the SP Copilot prototype, both as a technical artifact and as a workflow-support system for Ericsson's Solution Package process. The discussion focuses on the architectural choices made during the project, the shift away from vector-based RAG in the core workflows, the role of controlled tool integration, the value of human-in-the-loop governance, limitations of the current prototype, and possible future improvements.

7.1 Interpretation of the Proposed Architecture

The SP Copilot should be understood primarily as a hybrid enterprise workflow-support system rather than as a standalone chatbot or a conventional machine-learning model. This distinction is important. In many AI prototypes, the language model is placed at the center of the system and all other components are treated as supporting context providers. In this project, the opposite design logic is more appropriate: the workflow, enterprise systems, governance constraints, and user tasks define the architecture, while the LLM acts as one component inside a broader orchestration pipeline.

This design direction is consistent with Design Science Research, where the value of a system is judged not only by algorithmic novelty, but by how well the artifact addresses a practical organizational problem (Hevner et al., 2004; Peffers et al., 2007). In the SP workflow, the main problem is not a lack of generative text, but the difficulty of assembling, comparing, and interpreting distributed project information. Therefore, the system's contribution lies in the combination of data access, deterministic preprocessing, terminology grounding, rule-based analysis, constrained synthesis, and user-facing workflow control.

The architecture also reflects a broader movement in software engineering and project management toward integrated toolchains and workflow automation (Bass et al., 2015; Du-

mas et al., 2018; Weske, 2019; van der Aalst, 2016). However, unlike traditional automation systems that often depend on predefined process models, the SP Copilot must operate in a semi-structured environment. Confluence pages may contain a mix of templates, comments, checklists, tables, and historical notes. Jira issues provide structured execution data but may lack broader milestone meaning. FPT contributes planning context but is not always sufficient to explain operational readiness. The resulting information environment is therefore heterogeneous, incomplete, and partly informal.

For this reason, the prototype adopts a layered architecture. Low-level access is delegated to MCP-based tools and Python API facades, while high-level orchestration remains in Python. The Model Context Protocol provides a standardized way to encapsulate low-level operations against systems such as Confluence, Jira, and FPT (Anthropic, 2024; Model Context Protocol, 2025b,a). This separation improves maintainability because enterprise API details can be isolated from skill logic. It also creates clearer fault boundaries: if a specific connector fails, the application can report a controlled error or use Python fallback logic where available.

This approach is related to broader research on tool-using language models, where reasoning systems are combined with external actions and APIs (Yao et al., 2023; Qin et al., 2023; Schick et al., 2023; Nakano et al., 2021; Patil et al., 2023). However, the SP Copilot differs from many tool-agent systems because it does not give the model open-ended control over tools. Python orchestration decides which tools are called, when they are called, and how their outputs are assembled. This conservative integration model is better suited to governance-heavy enterprise environments, where tool use may expose permissions, operational records, and potential side effects.

7.2 From Vector RAG to Deterministic Grounding

One of the most important design changes during the project was the move away from vector-based RAG in the summary and risk workflows. RAG is a powerful and widely studied pattern for grounding language-model outputs in external knowledge (Lewis et al., 2020; Gao et al., 2024). Dense retrieval methods can be very effective when the relevant knowledge is expressed in semantically rich passages (Karpukhin et al., 2020; Reimers and Gurevych, 2019). However, the SP Copilot operates in a domain where many important terms are short acronyms, internal labels, gate names, and workflow-specific expressions. For these cases, semantic similarity is not always the most reliable retrieval signal.

In early prototypes, vector retrieval introduced two practical problems. The first was latency. Since the system is intended for interactive use by SP Drivers, long retrieval and reranking pipelines can reduce usability. The second was terminology risk. Acronyms may have several plausible meanings across a large enterprise. A semantically similar but contextually wrong glossary result can be more harmful than no result, because it may cause the model to build an explanation on an incorrect definition.

The final system therefore uses deterministic glossary indexing for terminology grounding. This approach is less flexible than semantic retrieval, but more appropriate for acronym and domain-term resolution. The tiered lookup strategy gives priority to the curated SP expert index before falling back to the general corporate glossary. This resembles classi-

cal information-retrieval principles, where exact matching, term weighting, and controlled vocabularies remain valuable when precision is more important than broad semantic recall (Spärck Jones, 1972; Salton and Buckley, 1988; Robertson and Zaragoza, 2009; Manning et al., 2008).

This does not mean that RAG is unsuitable for enterprise AI in general. Rather, the project shows that the choice of grounding method should depend on the type of knowledge being retrieved. For long procedural documents, local wiki lookup can benefit from lightweight lexical retrieval. For terminology, exact indexing is safer. For broad knowledge-intensive question answering, vector retrieval may still be useful. The main lesson is that grounding should be task-specific rather than treated as a single universal architecture.

7.3 Human-in-the-Loop Governance

The human-in-the-loop design is central to the system's practical usefulness. In enterprise workflows, automation is valuable only when users trust it and when accountability remains clear. Fully automatic updates to Jira or Confluence could create serious operational problems if the system misinterprets a page, applies a wrong mapping, or overwrites intentionally divergent information.

The synchronization skill therefore follows a proposal-before-execution pattern. It identifies possible discrepancies between Confluence-derived task structures and Jira issue states, but it does not execute changes until the user explicitly approves them. This reflects established principles in interactive machine learning and human-centered AI, where automation should support human judgment rather than bypass it (Amershi et al., 2014; Holzinger, 2016; Shneiderman, 2020). It also aligns with work on trust in automation, which emphasizes that users need appropriate reliance rather than blind acceptance (Lee and See, 2004; Parasuraman et al., 2000).

The same philosophy appears in the risk and prioritization skills. The risk skill does not claim to predict the future with statistical certainty. Instead, it surfaces evidence-backed concerns and explains why certain signals may matter. The prioritization skill does not directly change Jira priority values. It provides recommendations for SP Driver review. This preserves human accountability while still reducing the effort required to inspect large amounts of project data.

This design also helps manage the persuasive nature of LLM-generated text. Fluent output can appear more certain than the underlying evidence justifies. By keeping write actions behind explicit approval and by framing outputs as summaries, diagnoses, or recommendations rather than decisions, the system reduces the risk that users treat generated text as authoritative project truth.

7.4 Implications for the SP Workflow

From a workflow perspective, the prototype suggests several opportunities for improving the SP process.

First, SP coordination would benefit from stronger consistency between documentation and execution artifacts. The current need for synchronization support indicates that Con-

fluence and Jira often represent overlapping but not identical views of the project. Establishing clearer conventions for linking checklist items, milestone actions, and Jira issues would improve both human workflow and automated analysis. This relates to long-standing concerns in requirements and traceability research, where links between artifacts are essential for project understanding (Gotel and Finkelstein, 1994; Cleland-Huang et al., 2014; Mäder and Egyed, 2012).

Second, the SP workflow would benefit from more explicit evidence standards. Many project risks are not caused by missing work alone, but by unclear evidence: it may be difficult to determine whether a checklist item is complete, whether a comment is still current, or whether an old concern has been resolved. More standardized evidence fields, timestamps, and close-out markers would make both manual review and AI-assisted diagnosis more reliable.

Third, terminology management should be treated as infrastructure rather than documentation afterthought. The project showed that correct interpretation of acronyms is essential for reliable AI support. A curated SP-specific glossary, maintained together with process owners, can improve not only the copilot but also onboarding, reporting, and cross-team communication. This connects to broader knowledge-management theory, where organizational knowledge must be made explicit and reusable to support effective work (Nonaka and Takeuchi, 1995; Davenport and Prusak, 1998).

Fourth, the organization should distinguish between automation targets and decision-support targets. Some tasks, such as extracting task rows or detecting missing Jira links, are suitable for deterministic automation. Other tasks, such as judging whether a risk is serious, require contextual interpretation and human judgment. Treating all workflow problems as automation problems would be risky. A more sustainable approach is to automate repetitive context gathering while keeping judgment-rich decisions under human control.

7.5 Limitations of the Prototype

The current prototype has several limitations.

The first limitation concerns data completeness. The system can only reason over information it can access. If Jira links are missing, FPT data are unavailable, or Confluence pages contain outdated information, the copilot's output will be correspondingly limited. This is not only a technical issue, but also a workflow issue. AI support is strongest when source-system discipline is strong.

The second limitation concerns generalizability. The prototype is designed around Ericsson's SP workflow, its terminology, and its tool environment. Although the architectural principles may transfer to other enterprise settings, the concrete rules, glossary indexes, prompts, and synchronization logic would need adaptation.

The third limitation concerns evaluation. Since the system is a multi-skill enterprise artifact, it cannot be fully evaluated using a single benchmark metric. Functional correctness, latency, traceability, usefulness, and user trust must all be considered. This makes evaluation more complex than evaluating a standalone classifier or retrieval model. Methods from usability testing and human-centered AI evaluation are therefore relevant (Nielsen, 1993, 1994; Brooke, 1996; Amershi et al., 2019).

The fourth limitation concerns LLM reliability. Even with grounding and constrained

prompts, LLMs may still overgeneralize, omit relevant details, or phrase uncertain interpretations too strongly. Hallucination and faithfulness remain known challenges in language-model systems (Ji et al., 2023; Huang et al., 2024; Maynez et al., 2020). The prototype reduces these risks through evidence blocks, glossary lookup, and rule-based anchors, but it does not eliminate them completely.

The fifth limitation concerns deployment form. The prototype is implemented as a separate interface rather than as a fully integrated enterprise plugin. This is appropriate for research and experimentation, but it means that users must still switch context between the copilot and their ordinary working environment.

7.6 Future Optimization Directions

Several future improvements are possible. Some of these directions emerged from technical limitations observed during implementation, while others reflect feedback from SP Drivers and potential users of the prototype.

A first direction is to productize the prototype more tightly within the existing SP workflow. In its current form, the SP Copilot is presented through a separate Streamlit-based interface. While this is suitable for experimentation, SP Drivers indicated that a more convenient long-term form would be a tool embedded directly into the systems where they already work. In particular, a Confluence plugin or sidebar integration would reduce context switching and allow users to summarize pages, inspect risks, and trigger synchronization proposals directly from the relevant SP page.

A second direction is to extend the range of enterprise information sources. The current prototype focuses primarily on Confluence, Jira, FPT, glossary indexes, and the local wiki. However, important project knowledge may also exist in OneNote pages, email discussions, meeting notes, or other collaboration tools. Incorporating such sources could improve the completeness of the copilot's context, especially for decisions or concerns that are discussed informally before they are reflected in official project systems. This would require careful handling of access control, confidentiality, and source reliability, because not all informal communication should be treated as authoritative evidence.

A third direction is to support multimodal understanding. Some Confluence pages contain limited text but rely heavily on diagrams, flowcharts, architecture sketches, or embedded screenshots. In such cases, a text-only copilot may miss important structure. Future versions could include diagram analysis capabilities that extract process steps, dependencies, system components, or milestone flows from visual material. This would be especially useful when an SP page uses a diagram as the primary representation of project anatomy, dependency structure, or delivery flow.

A fourth direction is to improve traceability in generated outputs. Future versions could attach each factual claim to a source identifier, such as a Confluence section, Jira issue key, FPT field, wiki document, or diagram region. This would make outputs more auditable and help users distinguish between directly supported facts, weak evidence, and model interpretation. Stronger traceability would also reduce the risk of overtrusting fluent but insufficiently supported summaries.

A fifth direction is to strengthen evaluation with real user studies. SP Drivers could compare normal workflow execution against copilot-assisted execution on representative tasks.

Useful measurements could include time saved, number of missed issues, perceived trust, clarity of summaries, usefulness of risk diagnosis, and confidence in synchronization proposals. Such studies would provide stronger evidence of organizational value than prototype-level functional testing alone.

A sixth direction is to improve synchronization semantics and feedback learning. The current synchronization flow can detect mismatches and propose actions, but future work could introduce stronger identity resolution between Confluence checklist items and Jira issues. User feedback on accepted, rejected, or modified proposals could also be stored as a controlled signal for improving future recommendations. This would make the synchronization workflow more adaptive while still preserving explicit human approval.

Finally, future work could refine the rule engine and knowledge-maintenance process. Risk and prioritization rules could be calibrated through expert feedback and historical case analysis, while glossary and wiki knowledge should be maintained as living organizational assets rather than static files. In the long term, the copilot could become not only a consumer of enterprise knowledge, but also a mechanism for improving knowledge hygiene across the SP workflow.

7.7 Summary

The discussion highlights the central lesson of the project: enterprise AI systems require more than connecting an LLM to internal data. The difficult design work lies in deciding what data should be accessed, how it should be cleaned, which sources should be trusted, when deterministic logic should override generation, and where human approval is required.

The SP Copilot demonstrates a pragmatic pattern for enterprise AI: use protocols such as MCP for tool access, use deterministic methods for high-precision grounding, use rules for explicit diagnostic signals, use LLMs for synthesis and explanation, and use human-in-the-loop design for operational control. This combination is less autonomous than fully agentic systems, but it is more compatible with real organizational work, where usefulness, traceability, and accountability matter as much as generative capability.

Chapter 8
Conclusion

This thesis addressed the problem of information fragmentation and coordination overhead in Ericsson's Solution Package workflow. In this environment, project understanding depends on information distributed across Confluence, Jira, FPT, local knowledge sources, and domain-specific terminology. The practical burden of combining these sources falls largely on SP Drivers, whose work requires both operational follow-up and managerial interpretation.

To address this challenge, the thesis designed and analyzed the SP Copilot, presented through the SP Guardian interface. The system was implemented as a hybrid enterprise copilot combining Python-based orchestration, MCP-based tool integration, deterministic glossary indexing, rule-assisted analysis, constrained LLM synthesis, and human-in-the-loop action control. The prototype supports five main skills: multi-source summarization, risk-oriented diagnosis, synchronization, prioritization, and offline wiki lookup.

The work makes several contributions. First, it proposes an architecture for integrating Confluence, Jira, and FPT without forcing all data into one monolithic system. Second, it shows how MCP can be used as a low-level integration layer while leaving workflow orchestration in Python. Third, it demonstrates why deterministic terminology grounding can be preferable to vector-based RAG for acronym-heavy enterprise settings. Fourth, it implements a human-in-the-loop synchronization flow that preserves governance and user control. Fifth, it extends the copilot beyond summary and risk analysis by adding prioritization and wiki skills.

The main conclusion is that effective enterprise AI support requires more than connecting an LLM to internal data. In governance-heavy workflows, usefulness depends on how the system accesses data, how it grounds terminology, how it separates evidence from interpretation, how it controls write actions, and how it keeps humans responsible for consequential decisions. The SP Copilot therefore contributes not only a working prototype, but also a design perspective for building practical enterprise copilots.

The project also shows that the most valuable role for AI in this setting is not autonomous project management. Instead, the value lies in structured assistance: helping users gather

context faster, notice inconsistencies earlier, interpret risk signals more clearly, and reduce repetitive administrative work. This balance between automation and human control is likely to remain essential for future enterprise AI systems.

Future work should focus on productizing the prototype as a more integrated workflow tool, expanding the supported information sources, adding multimodal support for diagram-heavy pages, strengthening source-level traceability, improving synchronization identity matching, and conducting deeper user studies with SP Drivers. With these improvements, systems like SP Copilot could become an important support layer for complex industrial workflows, not by replacing human coordination, but by making that coordination more informed, consistent, and efficient.

References

Adadi, A. and Berrada, M. (2018). Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160.

Amershi, S., Cakmak, M., Knox, W. B., and Kulesza, T. (2014). Power to the people: The role of humans in interactive machine learning. *AI Magazine*, 35(4):105–120.

Amershi, S., Weld, D., Vorvoreanu, M., Fourney, A., Nushi, B., Collisson, P., Suh, J., Iqbal, S., Bennett, P. N., Inkpen, K., et al. (2019). Guidelines for human-ai interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13.

Anthropic (2024). Introducing the model context protocol. `https://www.anthropic.com/news/model-context-protocol`. Accessed: 2026-04-13.

Auth, G., Jokisch, O., and Dürk, C. (2019). Revisiting automated project management in the digital age – a survey of ai approaches. *Online Journal of Applied Knowledge Management*, 7(1):27–39.

Baeza-Yates, R. and Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology Behind Search*. Addison-Wesley, 2 edition.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Boehm, B. W. (1991). Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41.

Brooke, J. (1996). Sus: A quick and dirty usability scale. In Jordan, P. W., Thomas, B., McClelland, I. L., and Weerdmeester, B., editors, *Usability Evaluation in Industry*, pages 189–194. Taylor & Francis.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Cleland-Huang, J., Gotel, O., and Zisman, A. (2014). *Software and Systems Traceability*. Springer.

Davenport, T. H. and Prusak, L. (1998). *Working Knowledge: How Organizations Manage What They Know*. Harvard Business School Press.

de Bem, R. A., Farias, K., and de Souza, C. R. B. (2024). Large language models for project management: Opportunities, challenges, and research directions. *arXiv preprint arXiv:2402.18814*.

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.

Dumas, M., La Rosa, M., Mendling, J., and Reijers, H. A. (2018). *Fundamentals of Business Process Management*. Springer, 2 edition.

Endsley, M. R. (1995). Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64.

Fan, A. et al. (2023). A survey on large language models for software engineering. *arXiv preprint arXiv:2308.10620*.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., and Wang, H. (2024). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

Gotel, O. C. Z. and Finkelstein, A. C. W. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101.

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. (2019). A survey of methods for explaining black box models. *ACM Computing Surveys*, 51(5):1–42.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75–105.

Holzinger, A. (2016). Interactive machine learning for health informatics: When do we need the human-in-the-loop? *Brain Informatics*, 3(2):119–131.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Qin, B., and Liu, T. (2024). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*. Used for hallucination and grounding discussion.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547.

Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. (2020). Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6769–6781.

Kerzner, H. (2017). *Project Management Metrics, KPIs, and Dashboards: A Guide to Measuring and Monitoring Project Performance*. Wiley, 3 edition.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213.

Lee, J. D. and See, K. A. (2004). Trust in automation: Designing for appropriate reliance. *Human Factors*, 46(1):50–80.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Lin, J., Ma, X., Lin, S.-C., Yang, J.-H., Pradeep, R., and Nogueira, R. (2021). Pyserini: A python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2356–2362.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 30.

Mäder, P. and Egyed, A. (2012). Assessing the effect of requirements traceability for software maintenance. *IEEE Transactions on Software Engineering*, 38(3):609–626.

Malkov, Y. A. and Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Maynez, J., Narayan, S., Bohnet, B., and McDonald, R. (2020). On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919.

Model Context Protocol (2025a). Architecture overview. `https://modelcontextprotocol.io/docs/learn/architecture`. Accessed: 2026-04-13.

Model Context Protocol (2025b). Model context protocol specification. `https://modelcontextprotocol.io/specification/2025-06-18`. Version dated 2025-06-18; accessed 2026-04-13.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al. (2021). Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

National Institute of Standards and Technology (2023). Artificial intelligence risk management framework (ai rmf 1.0). `https://doi.org/10.6028/NIST.AI.100-1`.

Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufmann.

Nielsen, J. (1994). Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 152–158.

Nonaka, I. and Takeuchi, H. (1995). *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Parasuraman, R., Sheridan, T. B., and Wickens, C. D. (2000). A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 30(3):286–297.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77.

Project Management Institute (2021). *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Project Management Institute, 7 edition.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Lin, Y., Jiang, M., et al. (2023). Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. Technical report, OpenAI.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144.

Robertson, S. and Zaragoza, H. (2009). The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389.

Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Shneiderman, B. (2020). Human-centered artificial intelligence: Reliable, safe & trustworthy. *International Journal of Human–Computer Interaction*, 36(6):495–504.

Spärck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21.

van der Aalst, W. M. P. (2016). *Process Mining: Data Science in Action*. Springer, 2 edition.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Wallace, L., Keil, M., and Rai, A. (2004). How software project risk affects project performance: An investigation of the dimensions of risk and an exploratory model. *Decision Sciences*, 35(2):289–321.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

Weske, M. (2019). *Business Process Management: Concepts, Languages, Architectures*. Springer, 3 edition.

White, J., Hays, S., Fu, Q., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Yin, R. K. (2018). *Case Study Research and Applications: Design and Methods*. SAGE Publications, 6 edition.

Appendices

Appendix A
Formal Definitions of Core Workflows

This appendix provides compact formal definitions of several core operations in the SP Copilot. These definitions are not intended as theoretical contributions in themselves; rather, they are included to clarify the structure of the system's main workflows in a concise and implementation-independent form.

A.1 Integrated Information Space

The information space used by the copilot can be represented as:

$$D = D_c \cup D_j \cup D_f \cup D_g \cup D_w,$$

where D_c denotes Confluence-derived data, D_j denotes Jira-derived data, D_f denotes FPT-derived data, D_g denotes glossary entries, and D_w denotes documents in the offline wiki repository.

A.2 Tiered Glossary Lookup

The glossary space is divided into a domain-specific expert layer and a general organizational layer:

$$G = G_{\text{expert}} \cup G_{\text{general}}.$$

For a term t, the lookup function is defined as:

$$\text{lookup}(t) = \begin{cases} G_{\text{expert}}(t), & \text{if } t \in G_{\text{expert}}, \\ G_{\text{general}}(t), & \text{if } t \notin G_{\text{expert}} \wedge t \in G_{\text{general}}, \\ \emptyset, & \text{otherwise.} \end{cases}$$

This reflects the trust hierarchy used in the system, where expert SP terminology takes precedence over broader corporate terminology.

A.3 Synchronization Differential

Let A_c denote the set of normalized task attributes extracted from Confluence and A_j the corresponding set extracted from Jira. The system computes a discrepancy set:

$$\Delta = \text{diff}(A_c, A_j).$$

Given the human-in-the-loop design, only a user-approved subset

$$U \subseteq \Delta$$

may be executed. The updated system state is therefore represented as:

$$(A_c', A_j') = \text{apply}(A_c, A_j, U).$$

A.4 Prioritization Context Construction

For a set of Jira issues

$$J = \{j_1, j_2, \ldots, j_n\},$$

the prioritization module constructs a shared project context p and issue-specific contexts t_i:

$$t_i = \eta(j_i, p).$$

A deterministic heuristic score is then computed:

$$r_i = \rho(t_i).$$

The final recommendation is obtained by combining heuristic scoring with LLM-based synthesis:

$$y_i = \text{merge}(\text{LLM}(t_i, p), r_i).$$

Appendix B
Prompt Templates

This appendix documents the prompt-template design used in the final implementation of the SP Copilot system. Instead of relying on a single monolithic prompt, the implemented system adopts a modular prompting architecture. A dedicated router prompt first maps the user request to one of several specialized skills, and each skill then constructs its own task-specific prompt using structured runtime context such as Confluence page content, Jira data, FPT data, glossary context, and local wiki material.

The prompt design therefore follows a multi-stage orchestration pattern. This appendix summarizes the most important prompt categories used in the final system and explains their role in controlling skill selection, evidence grounding, output structure, and human-in-the-loop behavior.

B.1 Overview of the Prompting Architecture

The final prompting architecture consists of two stages.

1. **Routing stage.** A router prompt classifies the user request into one of the supported skills.
2. **Skill stage.** The selected skill builds a grounded prompt using task-specific evidence and output constraints.

In the current implementation, the router can dispatch to the following five skills:

- `summary`
- `risk`

- `sync`
- `priority`
- `wiki`

This design improves controllability compared with a single open-ended prompt. It also makes it easier to enforce different evidence hierarchies and output schemas for different tasks.

Table B.1: Main prompt categories in the implemented system

Prompt category	Main runtime context	Output format	Purpose
Router prompt	User request, conversation context, extracted identifiers, glossary-relevant terms	Strict JSON	Select skill, extract domain terms, rewrite the user query, and prepare normalized arguments.
Summary prompt	Confluence page text, checklist data, Jira context, FPT context, glossary entries, critical comment-description pairs	HTML	Produce factual SP summaries and page-aware targeted answers.
Risk prompt	Checklist evidence, expert-rule signals, Confluence evidence, Jira context, FPT context, glossary entries	HTML	Produce evidence-based risk diagnosis reports.
Priority context prompt	Confluence page context, milestone signals, page-level risks, completion evidence	Strict JSON	Infer project phase and compact page-level context for prioritization.
Priority recommendation prompt	Jira issue metadata, page context, milestone evidence, heuristic priority signals	Strict JSON	Recommend Jira task priority conservatively from structured evidence.
Wiki grounded QA prompt	Glossary context and retrieved local Markdown knowledge-base snippets	Text answer	Answer using only glossary and retrieved local wiki context.
Sync routing and review flow	User request, Confluence task structures, Jira issue structures, computed synchronization differences	Structured proposal view	Support human-reviewed synchronization proposals without direct autonomous writes.

B.2 Router Prompt

The router prompt is the first prompt executed in the runtime pipeline. Its task is not to answer the user directly, but to decide which skill should answer.

At runtime, the router prompt explicitly casts the model into the role of a *Router + Extractor + Query Rewriter + Tool-Argument Planner*. This is important because it shifts the model away from general chat behaviour and toward a controlled orchestration role.

The router prompt requests a strict JSON object containing:

- `skill`
- `vip_terms`
- `doc_search_query`
- `rewritten_user_query`
- `args`
- `needs_clarification`
- `clarification_question`

The prompt also contains several hard constraints. For example, it instructs the model to extract domain terms only from the current query text, not from URL parameters or page metadata, and to return only one skill from the allowed set. This output object is then consumed directly by the application runtime.

Conceptually, the router prompt performs four functions at once:

1. it classifies the user request,
2. it normalizes the query into a standalone form,
3. it extracts domain-specific entities such as gate tokens or acronyms,
4. it prepares structured arguments for downstream execution.

This is a typical example of control-oriented prompt engineering: the model is not asked for a natural-language answer, but for a machine-readable planning object.

B.3 Summary Prompt Template

The summary workflow uses one of the most complex prompt templates in the system because it must support both targeted page-aware questions and full SP page overviews.

The model is explicitly instructed to act as a factual reporter rather than a strategist. The prompt defines the model as a highly analytical SP data reporter whose task is to present objective facts from the available data sources. It is also explicitly told not to provide subjective risk ratings or advice.

The summary prompt is built around a structured hierarchy of truth. Different data blocks are designated as the primary source for different question types:

- `<fpt_data>` for schedule and date facts,
- `<jira_data>` for Jira issue counts and issue status,
- `<checklist_data>` for NOKs, completed items, and gate statistics,
- `<confluence_page>` for page wording, comments, and contextual explanation,
- `<critical_comment_description_pairs>` for structured evidence extracted from milestone tables.

This hierarchy reduces the risk of mixing incompatible evidence sources. For example, schedule questions should not be answered from free-text page comments when structured FPT schedule data is available.

A major domain-specific feature of the summary prompt is its explicit distinction between *template text* and *execution evidence*. Many SP pages contain generic lifecycle sections such as F1, F2, Verification, Delivery, FG, or FD, even if the SP has not yet reached those stages. The prompt therefore instructs the model not to treat generic lifecycle headings, definitions, or empty later-stage sections as proof of actual progression. Instead, it tells the model to prioritize dated logs, done tasks, open tasks, approvals, reviews, and concrete execution comments.

The summary prompt supports two top-level response scenarios:

1. **Scenario A:** targeted question or partial summary
2. **Scenario B:** comprehensive page summary or dashboard

Scenario B is further divided into three output paths:

- **Path A:** formal checklist or high-data gate page
- **Path B:** milestone or activity-tracking page
- **Path C:** general text or low-data page

Thus, the summary workflow is not driven by a single static prompt. It is a dynamic prompt template that adapts its output structure to the type of page and the user request. All summary outputs are constrained to simple HTML, which supports stable rendering in the frontend.

B.4 Risk Diagnosis Prompt Template

The risk workflow uses a diagnostic prompt template whose purpose is not simply to restate facts, but to interpret them as evidence-backed project risk.

The role definition in the prompt frames the model as a senior telecommunications project delivery expert and explicitly forbids unsupported claims such as invented delay days or risks not grounded in the input data.

Like the summary skill, the risk prompt injects multiple structured evidence blocks. However, the evidence hierarchy is different. The prompt explicitly prioritizes:

- `<checklist_evidence>`
- `<expert_engine_score>`
- `<critical_comment_description_pairs>`
- `<jira_defects>`
- `<fpt_schedule_data>`
- `<confluence_page>`

This difference reflects the analytical nature of the risk skill. In addition, the risk workflow is not purely LLM-based. It combines prompt-based reasoning with a rule-based expert layer that computes checklist severity and triggered risk categories before the prompt is sent to the model.

One of the strongest prompt patterns in this workflow is the embedded 2D risk matrix logic. The prompt instructs the model to evaluate each risk theme using:

- **Consequence** on a 1–5 scale,
- **Likelihood** on a 1–5 scale,
- **Risk level** derived from the likelihood–consequence combination.

The prompt also contains recency rules to distinguish current open risk from historical background. If a page contains late-stage approvals, close-out evidence, or completed delivery actions, the model is told not to escalate early-stage concerns unless the same issue is explicitly still open later in the page.

As with the summary skill, the risk skill supports different output paths:

- **Path A:** formal checklist or high-risk gate page
- **Path B:** milestone or activity-flow page
- **Path C:** general text or low-data page

The output is rendered as simple HTML. This keeps the frontend formatting predictable while allowing structured narrative reports.

B.5 Priority Recommendation Prompt Templates

The priority workflow uses two distinct prompt templates rather than one.

B.5.1 Project Context Extraction Prompt

The first prompt extracts a compact project context from the current Confluence page. Its purpose is to infer a concise state representation that can later guide Jira task prioritization.

The output schema includes fields such as:

- current phase,
- completion signal,
- short page context summary,
- active risks,
- supporting evidence.

This prompt is constrained to strict JSON. It also explicitly warns the model not to overfit to the beginning of the page, because SP pages may span multiple lifecycle stages and often contain historical material.

B.5.2 Priority Recommendation Prompt

The second prompt performs the actual Jira priority recommendation task. It instructs the model to recommend one of four priority levels:

- `Highest`
- `High`
- `Medium`
- `Low`

The prompt includes explicit mapping guidance. For example, `Highest` is reserved for items needing immediate attention, while `Low` is used for low-impact or already-completed work.

The model is explicitly forbidden from inventing missing fields such as due dates, dependencies, assignees, or comments. This is a conservative design choice because Jira data is often incomplete, and unsupported inference would otherwise inflate urgency.

The output is strict JSON and includes, for each issue:

- the issue key,
- the recommended priority,
- confidence,
- whether human review is required,
- a short justification,

- evidence,
- missing information affecting confidence.

This reveals an important architectural choice: the priority workflow is deliberately *read-only*. It does not directly update Jira priority values; instead, it provides a transparent recommendation for human review.

B.6 Wiki Grounded QA Prompt Template

The wiki workflow answers procedural and definitional questions from grounded local knowledge rather than from live project state.

Unlike the summary and risk skills, this workflow does not use live Confluence page evidence as its main knowledge source. Instead, it answers questions from two grounded context blocks:

- glossary context,
- retrieved local Markdown wiki context from `local_wiki/`.

The wiki prompt contains one of the strongest anti-hallucination instructions in the system. It explicitly states that the model must answer strictly from the provided glossary context and local wiki reference. If the term is not present, the model is told to say that the internal definition is not available rather than guessing from pre-trained knowledge.

In addition, the prompt requires source citation for claims grounded in the local wiki. This is especially important for internal enterprise knowledge support, where traceability is often more important than stylistic fluency.

Conceptually, the wiki prompt implements a tightly grounded retrieval-augmented answering pattern: retrieve local wiki snippets, prepend glossary context, and instruct the model to answer only from those blocks.

B.7 Interactive Sync and Move Flows

The synchronization and issue-move workflows are architecturally different from the other skills. In the current system, they are not primarily driven by one large free-form generative prompt in the same way as summary, risk, priority, or wiki answering.

Instead, the system uses the router prompt to enter the `sync` flow, and then relies on deterministic comparison logic plus an interactive review interface. Proposed Jira actions are displayed to the user, and no Jira write operation is executed until the user explicitly approves it.

This is a deliberate design decision. Because synchronization and issue movement can modify external systems, the project prioritizes auditability and human review over open-ended generation.

B.8 Cross-Cutting Prompt Design Patterns

Across the implemented prompts, several common prompt-engineering patterns can be observed.

B.8.1 Explicit Role Framing

Each prompt begins by assigning the model a narrow role, such as data reporter, risk expert, prioritization assistant, router, or internal knowledge assistant. This helps reduce generic assistant behaviour.

B.8.2 Structured Grounding Blocks

The prompts do not rely only on free-form instructions. They inject evidence into explicit blocks such as `<fpt_data>`, `<jira_data>`, `<checklist_data>`, and `<confluence_page>`. This makes the prompt behave more like a structured evidence assembly layer.

B.8.3 Anti-Hallucination Constraints

The prompts repeatedly forbid unsupported inference. Typical examples include:

- do not invent delay days,
- do not guess missing definitions,
- do not infer lifecycle stage from empty template headings,
- do not fabricate Jira or FPT data when those blocks are empty.

These constraints are necessary because SP pages often contain historical text, generic template sections, and incomplete supporting systems.

B.8.4 Strict Output Schemas

The prompts request either strict JSON or simple HTML. This reduces rendering ambiguity and supports deterministic runtime handling.

B.8.5 Dynamic Output Path Selection

The summary and risk skills both use conditional path selection depending on the structure of the page. This allows the same skill to handle checklist pages, milestone tables, and low-data pages without forcing one rigid format on all cases.

B.9 Prompt Design Discussion

The final system does not treat prompt engineering as the search for one universal instruction string. Instead, prompts are part of a broader orchestration design in which routing, evidence preparation, deterministic preprocessing, and constrained generation work together.

The router prompt decides what type of task the user is asking for, and each skill prompt defines what evidence hierarchy, reasoning pattern, and output schema should be used for that task. This design brings three major advantages.

First, it improves **control**. A fixed set of skills is easier to test and debug than one unrestricted prompt.

Second, it improves **grounding**. Different tasks rely on different evidence sources, and each prompt encodes that source hierarchy explicitly.

Third, it improves **frontend reliability**. Since outputs are constrained to JSON or simple HTML, they are easier to render consistently in the user interface.

For these reasons, the prompt templates in this project should not be understood only as pieces of text sent to a model. They are part of a larger runtime design that combines routing, structured retrieval, rule-based pre-processing, and carefully constrained generation.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-06-01

EXAMENSARBETE AI- and Automation-Driven Workflow Optimization for Product Delivery
STUDENTER Zhe Zhang, Biyu Zou
HANDLEDARE Anne Sjögren, Lifeng Han (Ericsson AB) och Pierre Nugues (LTH)
EXAMINATOR Görel Hedin (LTH)

När AI hjälper till att samordna produktleverans

POPULÄRVETENSKAPLIG SAMMANFATTNING **Zhe Zhang, Biyu Zou**

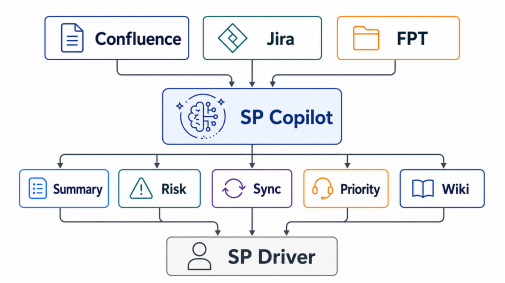
I stora utvecklings- och leveransprojekt finns viktig information ofta utspridd i flera system. Detta arbete undersöker hur AI och automation kan hjälpa projektkoordinatorer att snabbare förstå projektläget, upptäcka risker och minska manuellt dubbelarbete.

Moderna telekomprodukter utvecklas och levereras av många team samtidigt. I ett stort företag som Ericsson finns information om samma produktleverans ofta utspridd i flera system: dokumentation i Confluence, arbetsuppgifter i Jira och planeringsinformation i Feature & Product Tracker. Varje system fyller en viktig funktion, men tillsammans kan de göra det svårt att snabbt få en tydlig helhetsbild.

Detta examensarbete fokuserar på ett internt Ericsson-arbetsflöde för att samordna produkt- och funktionsleveranser. I detta sammanhang används begreppet Solution Package, eller SP, för en strukturerad leveransenhet som följer en produkt eller funktion genom olika milstolpar. Den som driver en SP behöver förstå vad som är klart, vad som saknas, vilka uppgifter som är blockerade och vilka risker som kan påverka leveransen.

I praktiken innebär detta mycket manuellt arbete. En projektkoordinator kan behöva läsa långa dokumentationssidor, kontrollera Jira-ärenden, jämföra datum, hitta bevis för milstolpar och tolka interna förkortningar. Utmaningen är därför inte bara att fatta rätt beslut, utan att först samla ihop rätt information från flera olika källor.

I projektet utvecklades en prototyp som kallas SP Copilot och visas genom gränssnittet SP Guardian. Systemet ersätter inte människor och fattar inte beslut automatiskt. I stället fungerar det som ett stödverktyg som samlar information, sammanfattar projektläget och hjälper användaren att se möjliga risker och prioriteringar.



Prototypen innehåller fem huvudfunktioner. Den kan skapa sammanfattningar från flera system, analysera risker, föreslå synkronisering mellan Confluence och Jira, ge rekommendationer om Jira-prioritering och besvara frågor från en lokal kunskapsbas. Viktigt är att ändringar i externa system inte sker automatiskt. Användaren måste först granska och godkänna föreslagna åtgärder.

En viktig lärdom är att AI i företagsmiljöer behöver vara mer än en språkmodell kopplad till data. Systemet måste veta vilka källor som är tillförlitliga, hur interna termer ska tolkas och när mänsklig kontroll krävs. Därför kombinerar SP Copilot regler, exakta uppslagningar av förkortningar, Model Context Protocol för verktygsintegration och språkmodeller för tydliga sammanfattningar.

Genom att minska tiden som läggs på informationssökning kan SP Copilot hjälpa projektkoordinatorer att arbeta mer effektivt, upptäcka problem tidigare och behålla kontrollen över viktiga beslut.