UNIVERZA V LJUBLJANI

FGG Fakulteta za gradbeništvo in geodezijo

SANGEEN KHAN

GENERATIVE DESIGN PROPOSALS VIA LLM-BIM INTEGRATION

PREDLOGI GENERATIVNEGA OBLIKOVANJA PREKO INTEGRACIJE LLM-BIM

Master thesis No.:

Supervisor:
Prof. Žiga Turk, Ph.D.

President of the Committee
Prof. Goran Turk, Ph.D.

Ljubljana, 2025

ERRATA

Page	Line	Error	Correction

BIBLIOGRAFSKO – DOKUMENTACIJSKA STRAN IN IZVLEČEK

UDK: 004.42:69(043.2)

Avtor: Sangeen Khan

Mentor: Prof. Dr. Žiga Turk

Somentor:

Naslov: Predlogi generativnega oblikovanja preko integracije LLM-BIM

Tip dokumenta: magistrsko delo

Obseg in oprema: **57str., 14 sl.**

Ključne besede: Model Context Protocol, povpraševanja v naravnem jeziku, informacijsko modeliranje stavb, integracija AI-BIM, dvosmerna komunikacija, pogovorna vmesnika

Izvleček: Veliki jezikovni modeli (LLM) imajo edinstvene prednosti in potencial v okoljih informacijskega modeliranja gradenj (BIM). Njihova metodologija podpira interakcijo v naravnem jeziku s kompleksno programsko opremo za načrtovanje, kar omogoča učinkovito komunikacijo med sistemi umetne inteligence in platformami, ki temeljijo na BIM-u. Vendar pa za delovne tokove strukturnega in arhitekturnega modeliranja integracijska zmogljivost obstoječih pristopov AI-BIM predstavlja nekatere omejitve, ki jih je treba rešiti z najboljšimi možnimi rešitvami. Ti izzivi vključujejo enosmerne komunikacijske vzorce in kompleksne programske zahteve, ki ovirajo implementacijo pogovornih delovnih tokov umetne inteligence v BIM načrtovalskih pisarnah.

Ta študija se osredotoča na analizo procesa omogočanja dvosmerne komunikacije med LLM-i in BIM programsko opremo, hkrati pa razvija edinstven delovni tok za implementacijo znotraj načrtovalskih orodij, platform umetne inteligence in projektnih ekip. Razviti sta bili dve študiji primerov za testiranje praktičnih interoperabilnostnih delovnih tokov: BIMCP za integracijo z Autodesk Revitom in Kolektor-BONCP za manipulacijo IFC preko Blenderjevega dodatka Bonsai. Namen je identificirati in rešiti omejitve, povezane s procesiranjem ukazov v naravnem jeziku in pretokom podatkov med asistenti umetne inteligence in BIM programsko opremo. Komunikacijski procesi so dvosmerni, z uporabo ogrodja Model Context Protocol (MCP) in povezav, ki temeljijo na vtičnicah, za interakcijo v realnem času.

Integracija AI-BIM in pogovorni načrtovalski delovni tokovi so relativno premalo izkoriščeni. To je predvsem posledica problemov z interoperabilnostjo in ovir pri dostopnosti. Kljub identificiranim omejitvam in tehničnim izzivom rezultati študije znotraj razvoja študij primerov prikazujejo, kako ima delovni tok odlične prednosti pri razvoju projektov BIM s pomočjo umetne inteligence in njegovi implementaciji v načrtovalskih pisarnah gradbene industrije. Validacija v realnem svetu pri podjetju Kolektor Koling dokazuje merljive izboljšave produktivnosti, saj pretvarja večurne ročne procese v minutno dolge pogovorne poizvedbe.

BIBLIOGRAPHIC– DOKUMENTALISTIC INFORMATION AND ABSTRACT

UDC:	004.42:69(043.2)
Author:	Sangeen Khan
Supervisor:	Prof. Žiga Turk, Ph.D.
Cosupervisor:	
Title:	Generative Design Proposals via LLM-BIM Integration
Document type:	Master Thesis
Scope and tools:	**57p., 14 fig.**
Keywords:	Model Context Protocol, Natural Language Prompts, Building information Modeling**,** AI-BIM Integration, Bidirectional Communication, Conversational Interfaces

Abstract:

Large Language Models (LLMs) have unique advantages and potential in Building Information Modeling (BIM) environments. Their methodology supports natural language interaction with complex design software, offering efficient communication between AI systems and BIM-based platforms. However, for structural and architectural modeling workflows, the integration capacity of existing AI-BIM approaches presents some limitations that must be resolved with best possible solutions. These challenges include unidirectional communication patterns and complex programming requirements that hinder implementing conversational AI workflows in BIM design offices.

This study focuses to analyze the process of enabling bidirectional communication between LLMs and BIM software while developing a unique workflow for implementation within design tools, AI platforms, and project teams. Two case studies have been developed to test practical interoperability workflows: BIMCP for Autodesk Revit integration and Kolektor-BONCP for IFC manipulation through Blender's Bonsai addon. This is to identify and resolve the limitations involved in natural language command processing and data flow between AI assistants and the BIM software. The communication processes are bidirectional, using the Model Context Protocol (MCP) framework and socket-based connections for real-time interaction.

AI-BIM integration and conversational design workflows are relatively underutilized. This is mainly due to interoperability problems and accessibility barriers. Despite the identified limitations and technical challenges, the study outcomes within the case study development illustrate how the workflow has excellent advantages in developing AI-assisted BIM projects and its implementation in construction industry design offices. Real-world validation at Kolektor Koling demonstrates measurable productivity improvements, transforming multi-hour manual processes into minute-long conversational queries.

ACKNOWLEDGEMENTS

First and foremost, I thank Allah for giving me the strength and knowledge to complete this journey.

This thesis is dedicated to my daughter, Rameen Khan, my greatest joy and inspiration.

I am deeply grateful to my supervisor, Prof. Žiga Turk, a truly fantastic person who guided me from start to finish. He polished my skills in BIM and research, and his help connecting me with construction and IT companies made all the difference. His support went far beyond academic supervision.

To my father, mother and my beloved wife and my brothers – your financial support and belief in me made this dream possible. Thank you for standing by me through every challenge.

Thank you to the BIM A+ European Master program for this opportunity to advance in Building Information Modelling, and to Kolektor Koling for giving me a platform to work on real projects and explore new tools in construction and AI.

My sincere thanks to Dr. Sajjad Wali Khan and Dr. Faisal Javed for their continuous support in helping me reach this level. I also thank Metod Gaber and Dejan Papič of Kolektor Koling for their active support and participation throughout this research.

To my dear friends – Mohsin Ali Khan, thank you for always being there with moral support, and Afonso Ramos Portela, for your technical help when I needed it most. I am also grateful to Babar Ilyas, Muhammad Awais, Saghir Amin, Maria Clara, Bakht Yaseen, Ahtisham Ali Baig, Ashiqul Mursalin Choudary, and Haris Waheed Bhatti for their help and support throughout my studies and life.

TABLE OF CONTENTS

ERRATA .... **I**

BIBLIOGRAFSKO – DOKUMENTACIJSKA STRAN IN IZVLEČEK .... **IV**

BIBLIOGRAPHIC– DOKUMENTALISTIC INFORMATION AND ABSTRACT .... **VI**

ACKNOWLEDGEMENTS .... **VIII**

TABLE OF CONTENTS .... **IX**

INDEX OF FIGURES .... **XII**

1 INTRODUCTION **1**

1.1 BACKGROUND: 1

1.2 PROBLEM STATEMENT 2

1.3 RESEARCH QUESTIONS AND OBJECTIVES 3

1.3.1 Primary Research Question 3

1.3.2 Research Objectives 3

1.4 THESIS STRUCTURE OVERVIEW 3

Chapter 2: Literature Review 3

Chapter 3: LLM-BIM System Architecture 3

Chapter 4: Co-pilots For BIM Softwares 4

Chapter 5: SWOT Analysis 4

Chapter 6: Conclusion 4

2 LITERATURE REVIEW **5**

2.1 Frameworks for LLM-BIM Communication 5

2.2 Translating subjective natural language into objective BIM operations 5

2.3 Technical Constraints in BIM Ecosystems 6

3 LLM-BIM SYSTEM ARCHITECTURE **8**

3.1 COMPONENTS 8

3.1.1 User 8

3.1.2 Large Language Model 8

3.1.3 BIM Model ... 9
3.1.4 Software Tools ... 9
3.1.5 Connection between LLM and Softwares ... 10
3.1.6 Model Context Protocol ... 10
4 CO PILOTS FOR BIM SOFTWARES ... 16
4.1 BIMCP ... 16
4.1.1 System Architecture Overview ... 16
4.1.2 Communication Workflow ... 33
4.1.3 Testing and Debugging Framework ... 38
4.2 KOLEKTOR-BONCP ... 42
4.2.1 System Architecture ... 43
4.2.2 Tools ... 43
4.2.3 Data Export and Analysis Capabilities ... 44
4.2.4 Visual Feedback and Model Manipulation ... 45
4.2.5 Sequential Thinking Integration ... 46
4.2.6 Docker Deployment and Enterprise Integration ... 46
4.2.7 Practical Business Impact ... 47
4.2.8 Comparison with BIMCP ... 47
5 SWOT ANALYSIS ... 49
5.1 STRENGTHS ... 49
5.2 WEAKNESSES ... 49
5.3 OPPORTUNITIES ... 50
5.4 THREATS ... 51
5.5 STRATEGIC ASSESSMENT ... 51
6 CONCLUSION ... 52
6.1 SUMMARY OF RESEARCH ACHIEVEMENTS ... 52
6.2 ADDRESSING THE RESEARCH QUESTIONS ... 52
6.2.1 Primary Research Question ... 52
6.2.2 Subsidiary Research Questions ... 53

6.3 CONTRIBUTIONS TO AEC INDUSTRY 53

6.4 TECHNICAL ACHIEVEMENTS AND INNOVATION 54

6.5 FINAL REMARKS.. 54

7 REFERENCES 56

INDEX OF FIGURES

Figure 3-1 Traditional Workflow and Advanced Workflow with Natural Language Input.................... 8
Figure 3-2 Integration of Tools with LLM without MCP and with MCP.. 11
Figure 3-3 Capabilities of MCP Server.. 13
Figure 4-1 Tradional and Natural language wrokflow in Revit 16
Figure 4-2 Integration of softwares with MCP 17
Figure 4-3 Resources developed in BIMCP.. 20
Figure 4-4 Initiliztion of Revit plugin 32
Figure 4-5 Claude to Revit request through natural language prompt 34
Figure 4-6 Response from Revit to Claude in JSON format.. 35
Figure 4-7 Display of information by Claude in natural language.. 36
Figure 4-8 Sequence diagram of Request-Response from Claude to Revit and vice versa 37
Figure 4-9 Initialization of MCP inspector 38
Figure 4-10 Inspection of (a) Resources (b) Prompts and (c) Tools 40
Figure 4-11Tradional and Natural language wrokflow in Bonsai.. 42

1 INTRODUCTION

1.1 BACKGROUND:

Computers are invented to make human life easy but at the same time advanced IT systems are used to make it easy for stupid computers to understand. Computers always need to be spoon-fed with structured data, the data which is organized in a predefined manner, such as databases or spreadsheets or libraries inside tools like Revit etc, while humans always have the capability to work with unstructured data. At the same time, in the built environment, there are AI based technologies like self-driving cars which work on unstructured data. It leads to the question that how to have both worlds: humans using common sense and natural language concepts but resulting information is transformed into structured data for software to analyze it?

In BIM, to transform the unstructured data (natural language prompt) to structured data (Complex BIM Models), machine learning and artificial intelligence techniques like neural networks and computer vision integration is a hope but it also has some challenges. First challenge is that large datasets need to be trained on AI to generate a perfect model inside softwares which is based on thousands of training data and a lot of time to train and test these models. The second challenge is the BIM software for which the data is trained as each software has different architecture and technical constraints due to which changes in that software are difficult and exhaustive.

To solve this problem, researchers started using Large Language Models (LLMs) instead of traditional machine learning. The good thing about LLMs is that these models already know how to understand language - there is no need to train them with thousands of BIM examples like other AI systems require. Models like GPT-4 and Claude can understand what architects mean when something is said in plain English and then convert it to BIM commands. This is much better than the old way where separate training was needed for each BIM software. Now researchers can quickly make systems where user can talk to BIM software using normal language but most of these approaches remain limited to unidirectional information flow.

Current implementations can generate 3D building models from natural language descriptions or automate specific tasks like wall detailing, but they cannot query users for clarification when instructions are ambiguous or report back when operations fail. These systems operate as one-way pipelines that process commands without validating whether the outputs actually meet the user's intent. Even voice-based interfaces, despite their promise of natural interaction, still create steep learning curves and heavy cognitive loads because users receive no feedback when commands are misinterpreted or fail to execute. The fundamental issue is that these interfaces lack mechanisms for the BIM system to request clarification or provide contextual feedback about the current model state. While LLMs can successfully

translate human language into BIM operations, the absence of bidirectional communication prevents the truly conversational interaction needed to bridge the human-computer divide.

This unidirectional limitation becomes more problematic when considering the complex nature of architectural design tasks. Current systems process each command in isolation without understanding what was done before or maintaining any memory of previous operations. If an architect says, "make that wall taller," existing systems fail because they have no concept of "that wall" - they don't remember which wall was just created or modified. Similarly, when operations fail due to BIM's internal constraints like invalid geometry or missing prerequisites, these systems cannot explain what went wrong or suggest alternatives in human-understandable language. The absence of true dialogue means architects must carefully structure their requests to avoid ambiguity and manually verify results after each operation.

The lack of visual understanding further compounds these issues, as the systems operate blindly without knowing the current state of the model. An architect might unknowingly request impossible operations like "add a door to the curved wall" without the system being able to see that the wall's geometry doesn't support door placement. Additionally, existing implementations hard-code specific BIM operations, making them inflexible when new design requirements emerge. These fundamental limitations mean that despite the promise of natural language interaction, current LLM-BIM integrations still require users to think like programmers rather than designers. The disconnect between how architects naturally conceptualize designs and the rigid, sequential manner in which these systems operate undermines the very purpose of natural language interfaces - to make BIM software more intuitive and accessible.

The integration of Large Language Models (LLMs) with Building Information Modeling (BIM) environments represents an emerging frontier in architectural design automation. As the construction industry increasingly adopts digital workflows, researchers are exploring how natural language interfaces can make BIM systems more accessible and powerful. This review examines current approaches to LLM-BIM integration, focusing on technical architectures, semantic translation challenges, and implementation constraints.

1.2 PROBLEM STATEMENT

While the background shows that LLMs could solve BIM's complexity problem, there is no complete framework for creating real two-way communication between natural language interfaces and BIM software. Without this framework, the construction industry cannot use AI to its full potential because current solutions only fix small parts of the problem. This missing piece stops us from building systems that could change BIM from something hard to use into something that helps everyone adopt digital tools in construction. As a result, the industry cannot use advanced technology to work faster and design better buildings.

1.3 RESEARCH QUESTIONS AND OBJECTIVES

1.3.1 Primary Research Question

How can Large Language model be integrated with Building Information Modeling environments to enable direct, bidirectional interaction that assists architects, engineers and construction industry professionals?

Subsidiary Research Questions

RQ1: What technical architectures enable reliable communication between AI assistants and BIM authoring tools?

RQ2: How effectively can natural language commands express complex BIM operations compared to traditional programming approaches, and what are the semantic challenges in translating design intent?

RQ3: What are the technical constraints when implementing AI-driven BIM automation within existing software ecosystems, and how can these limitations be addressed?

1.3.2 Research Objectives

The objective of this research is to find the answers to these questions through studying the literature and to explore new ways of development for AI-BIM integration.

1.4 THESIS STRUCTURE OVERVIEW

This thesis presents a systematic journey from theoretical foundations through practical implementation, demonstrating how natural language interfaces can enable AI-BIM integration through the Model Context Protocol framework.

Chapter 2: Literature Review examines current approaches to LLM-BIM integration, analyzing frameworks for natural language communication with BIM systems, semantic translation challenges, and technical constraints within existing software ecosystems. The review establishes gaps in current integration approaches and positions Model Context Protocol as an enabling technology for bidirectional AI-BIM interaction.

Chapter 3: LLM-BIM System Architecture provides comprehensive coverage of the Model Context Protocol, detailing its components, system architecture, protocol design, and core concepts including resources, prompts, tools, sampling, roots, and transports. This chapter establishes the theoretical and technical foundation for implementing AI-BIM integration through standardized communication protocols.

Chapter 4: Co-pilots For BIM Softwares presents two practical implementations of MCP-based systems. BIMCP (Building Information Model Context Protocol) demonstrates integration with Autodesk Revit through dynamic C# code execution, comprehensive resource systems, and socket-based communication. Kolektor-BONCP showcases IFC manipulation through Blender's Bonsai addon, featuring specialized tools for data extraction, visual feedback capabilities, and Docker deployment for enterprise integration. Both implementations validate the MCP approach through real-world applications.

Chapter 5: SWOT Analysis provides a strategic assessment of the developed systems, analyzing strengths such as bidirectional communication and productivity improvements, weaknesses including platform dependencies and cost constraints, opportunities for expansion to additional BIM platforms and integration modalities, and threats from rapidly evolving AI technologies and industry resistance.

Chapter 6: Conclusion synthesizes research achievements, systematically addresses the research questions, documents contributions to the AEC industry The chapter emphasizes how natural language interfaces can democratize access to advanced BIM capabilities while establishing foundations for continued innovation in AI-assisted construction workflows.

The thesis demonstrates that natural language interfaces become practical when systems prioritize accessibility alongside technical capability. Both BIMCP and Kolektor-BONCP prove that conversational AI can serve as the universal interface enabling construction professionals to harness the full potential of digital building information systems.

2 LITERATURE REVIEW

Going through the landscape of papers published on this topic, three core research areas that are dominating the field are discussed here

2.1 FRAMEWORKS FOR LLM-BIM COMMUNICATION

Recent research has established several promising frameworks for enabling communication between LLMs and BIM platforms.[1] introduced a six-step LLM-augmented BIM framework (interpret-fill-match-structure-execute-check) demonstrated through NADIA-S, a speech-to-BIM application that addresses the cognitive burden of remembering complex command sequences. Building on this,[2] developed Text2BIM, a multi-agent framework where LLMs collaborate to generate executable code that produces native BIM models. Their approach represents building models as imperative code scripts invoking BIM APIs, enabling iterative improvement through feedback loops and domain-specific rule checking.

2.2 TRANSLATING SUBJECTIVE NATURAL LANGUAGE INTO OBJECTIVE BIM OPERATIONS

[3]introduced an intelligent retrieval engine that leverages natural language processing (NLP) techniques to facilitate user-friendly access to BIM object databases. Their system interprets user queries in natural language and retrieves relevant BIM objects, focusing on improving accessibility for non-expert users and streamlining the search process within BIM platforms. Building on this initial work,[4] advanced the field by developing an ontology-aided semantic parser capable of handling multi-constraint queries. Their approach introduces a modular ontology that maps natural language expressions to Industry Foundation Classes (IFC) concepts and relationships, enabling the system to process complex queries involving multiple attributes and relational constraints. The key relation between the two studies lies in their shared goal of bridging the gap between human language and BIM data retrieval; Yin et al. extend the capabilities explored by Wu et al. by incorporating ontological reasoning and semantic parsing, allowing for more precise and fine-grained information retrieval from BIM models.

Similarly,[5] introduced NADIA, which implements strategic separation between specification and creation of building elements, using appropriate prompting to guide LLMs toward architecturally rational responses while maintaining seamless BIM-LLM chaining.

Understanding user intent and extracting relevant parameters from natural language remains a critical challenge.[6] proposed T2S4BIM, a text-to-structure approach that extracts both user intent (operation type) and slots (targeted elements and properties) from natural language requests. Their research demonstrated that encoder-decoder models like T5 achieved performance comparable to larger decoder-

only models while improving efficiency. This work highlights the importance of synthetic data generation for training domain-specific language understanding systems.

Other studies detail approaches that allow speech interaction with the design tools. For example,[7] and [8], developed a framework that requires no knowledge of BIM commands, allows natural language voice input for searching and manipulating BIM data directly in software. This could significantly lower the learning curve of new users interacting with complex BIM software.

Two comprehensive meta-analyses, by [9] and [10], offer a detailed portrait of how natural language processing (NLP) and text mining are transforming the construction industry, particularly in their integration with BIM. Together, these studies map the evolution, research trends, and practical applications of NLP and text mining, highlighting the sector's shift towards managing unstructured data and advancing digitalization.[9] analyzes 254 papers from the Scopus database (1989–2020), using science mapping tools (BiblioShiny, VosViewer, Gephi) to visualize research trends, conceptual structures, and collaboration networks which revealed that NLP-BIM integration has grown steadily, with notable increases since 2015, and specific core research in topics like automated compliance checking and semantic BIM enrichment, two trends that reveal the importance of managing unstructured data in construction.[10] reviewed 205 articles focusing on the applications of text mining and natural language processing (NLP) within the construction industry. The authors systematically analyse how these technologies are being leveraged to address the sector's unique challenges, such as managing vast amounts of unstructured textual data from documents, reports, and communications. The review highlights key application areas including automated document classification, information extraction, sentiment analysis, compliance checking, and knowledge management. It also discusses the evolution of methods from traditional rule-based and statistical approaches to more advanced machine learning and deep learning techniques.

These analysis present major research trends, technological gaps, and future opportunities, emphasizing the growing importance of NLP and text mining in enhancing digitalization, efficiency, and decision-making in construction projects.

2.3 TECHNICAL CONSTRAINTS IN BIM ECOSYSTEMS

The practical applications of LLM-BIM integration span the entire building lifecycle. In design phases, systems enable automated detailing, space planning, and code compliance checking through natural language queries.[11] demonstrated autonomous construction robot control through LLM communication, successfully handling complex tasks like material stacking and pipeline installation. For construction management, transformer language models map schedule activities to standard categories, enabling intuitive project interfaces [5]. These applications extend to facility management, where natural language interfaces facilitate maintenance requests and energy performance queries.

Despite these advances, significant gaps remain. The lack of standardized communication protocols between AI systems and BIM platforms creates integration challenges across different software ecosystems. Current implementations primarily focus on command execution rather than design reasoning, suggesting opportunities for systems that can criticize designs, explain decisions, and propose alternatives. The success of multi-agent approaches indicates potential for more sophisticated collaborative systems with specialized agents for different design disciplines.

The emergence of Model Context Protocol and similar standardized approaches offers a path forward for achieving truly bidirectional interaction between LLMs and BIM systems. While academic research on MCP-BIM integration remains limited, [12] recently spoke a conceptual multi-agent system framework that leverages MCP and Agent-to-Agent (A2A) protocols for engineering design workflows. This framework addresses the fragmentation across structural, geotechnical, fluid dynamics, and BIM tools by enabling agents to interpret and utilize tool functionalities without explicit programming. The approach shifts from traditional linear processes toward decentralized, intelligence-driven paradigms, supporting use cases such as parametric design optimization and automated BIM synchronization. However, Ghosh identifies significant challenges including state management, security validation, and legacy API interoperability that must be addressed in practical implementations.

As demonstrated by recent implementations, these protocols can handle the complexities of BIM database transactions, context preservation across interactions, and thread-safe operations. The integration of LLMs with BIM represents not merely a technical advancement but a fundamental shift in how designers interact with digital building models, promising to enhance design quality, productivity, and accessibility.

3 LLM-BIM SYSTEM ARCHITECTURE

3.1 COMPONENTS

The architecture of LLM-BIM integration contains the apart from BIM software and LLM, User, BIM Model and connection between the LLM and BIM Software which in this case is the socket connection and model context protocol. Each one of these is discussed below in detail.

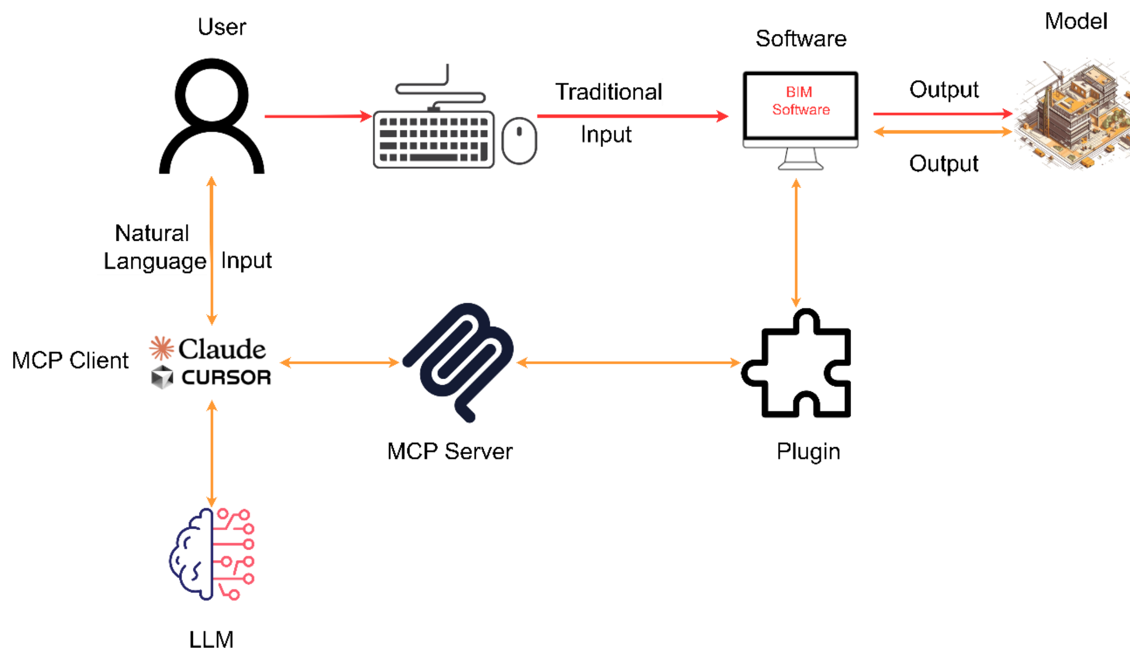


Figure 3-1 Traditional Workflow and Advanced Workflow with Natural Language Input

3.1.1 User

The user is the human which can be from Architecture Engineering, Construction and Operation (AECO) industry or it can be any stakeholder in the BIM industry, for example owner or client of the project. In this research the users are architects, engineers, managers, cost estimators and sales persons.

3.1.2 Large Language Model

As mentioned in the chapter 1 the large language model (LLM) is a type of artificial intelligence model which is trained on human written text data to learn the pattern and then translate and generate text based on that pattern. Initially it was trained for plain text now various companies are producing their models for coding and programming stuff and for image translation and generation.

Some LLMs are open source and freely available for developers to use and modify, while others are proprietary and require paid access through APIs. Generally, larger and more sophisticated models tend to perform better but come with higher computational costs and pricing. The effectiveness of these

models often correlates with their size, training data quality, and the resources invested in their development, making premium models typically more capable than their free counterparts.

Recent examples include DeepSeek-R1, which had its full public release in January 2025 as an improved open-source reasoning model that can compete with OpenAI's expensive o1 model while costing much less to develop and run, with continued updates like the R1-0528 version released in May 2025. There's also GPT-4o from OpenAI, which handles text, images, and voice all together with fast response times. Anthropic offers Claude 4 with different versions like Sonnet 4 and Opus 4 that are specifically built to be safe and reliable for businesses to use.

From China, we've seen Qwen 3 from Alibaba released in April 2025, which performs well across various tasks. Google's Gemini 2.5 Pro can process enormous amounts of text at once - up to 1 million tokens, which means it can handle entire books or lengthy documents in a single conversation.

On the open-source side, companies like Mistral provide models that researchers and businesses can freely use and modify, while IBM's Granite family offers completely open-source options under permissive licenses. This mix of free and paid options gives developers and companies flexibility to choose what works best for their needs and budgets.

Open source models have also some constraints for example the best models which are trained on more data, requires more computation power which normal computer don’t have so for this reason in this research, Claude LLM is used with paid subscription and it is discussed in the coming section of model context protocol that why Claude models were chosen and not the others.

3.1.3 BIM Model

The name indicates, this is a detail digital model of an actual physical structure in the form of 3D geometric elements along with meta data like material properties etc about the geometry in proprietary software format like .rvt etc. or Industry Foundation Class (IFC) .ifc format.

Unlike 2d format the BIM model is intelligent model which contains a database of information which helps professionals to analyze and visualize the projects throughout their lifecycle from planning and design to construction and operation and maintenance.

3.1.4 Software Tools

For creating these BIM models several BIM softwares are used. BIM software is specialized computer programs designed to create, edit, and manage digital building models and infrastructure projects. These applications allow users to design buildings, roads, bridges, tunnels, and other infrastructure in 3D while automatically generating 2D drawings, schedules, and reports from the same model. The software integrates architectural, structural, MEP (mechanical, electrical, plumbing), and civil engineering

systems into a single coordinated model that multiple team members can work on simultaneously. Famous BIM software includes Autodesk Revit for general building design, Graphisoft ArchiCAD for architecture, Tekla Structures for structural engineering, Bentley MicroStation for infrastructure projects, and Civil 3D for roads, highways, and site development.

3.1.5 Connection between LLM and Softwares

Here comes the important thing in the LLM-BIM integration workflow. BIM software primarily connects to LLMs using HTTP REST API calls through native add-ins or plugins. For Revit, this typically involves C# .NET add-ins that make direct HTTP requests to services like OpenAI's ChatGPT API using the standard HttpClient class with API key authentication. Other BIM software like AutoCAD and ArchiCAD use similar approaches with their respective plugin architectures making JSON-based REST calls to LLM services. This direct API integration pattern is more widely adopted than complex multi-language frameworks, with MCP being a newer emerging standard that aims to simplify these connections further.

3.1.6 Model Context Protocol

MCP originated as part of internal project by anthropic initially intended to improve the performance of Claude Desktop so that it can interact with local file system and other external systems like google drive, Gmail, calendar and github etc. It was then made open source so that every ai application can be connected for better performance and to get more benefits. Anthropic published its specifications to make it available for developers.

The MCP ecosystem consists of a growing number of mcp servers which are created by Anthropic MCP team and the open source community. MCP is model agnostic, and it is possible to hooked into different language models. Since it is open source so everyone can make mcp server and connect it to their application.

MCP is very important because to give context to LLM models makes the model perform in the best way. This is what mcp is doing. It is basically a protocol written to give context to any large language model to perform action on external applications, that’s why it is named as *Model Context Protocol*.

There can be a very intelligent model in the front but if it is not connected well or if it is not connected to application and it cannot get context and live information from the application, it will not perform well. On the other hand, if the model is even weak but if it is given context, it performs well so that’s why the importance of mcp in this regard cannot be denied.

Here a question raises that “Everything which is done with mcp, can be done without mcp?” The answer is yes, it can be done without mcp but the problem is that as it is discussed previously about MXN

problem which means that every large language model will need a separate protocol for every application which limits its functionality and makes it more tedious for doing it for every model and application separately. The problem even becomes more complex when new models are developed, and new applications and data sources are produced. This repetitive process becomes exhausting and time consuming ultimately leads to loss in productivity and interest.

The model context protocol borrows a lot of its ideas from other protocols which were developed for the same kind of idea, for example the Language server protocol or LSP. It was developed by Microsoft in 2016 which standardized how integrated development environments interact with language specific tools. When extensions are created for particular languages and particular development environments, the protocol is not written over and over again for all those development environments. Yet having same idea but MCP is novel and a bit advance in standardizing the interaction between ai application and external systems.

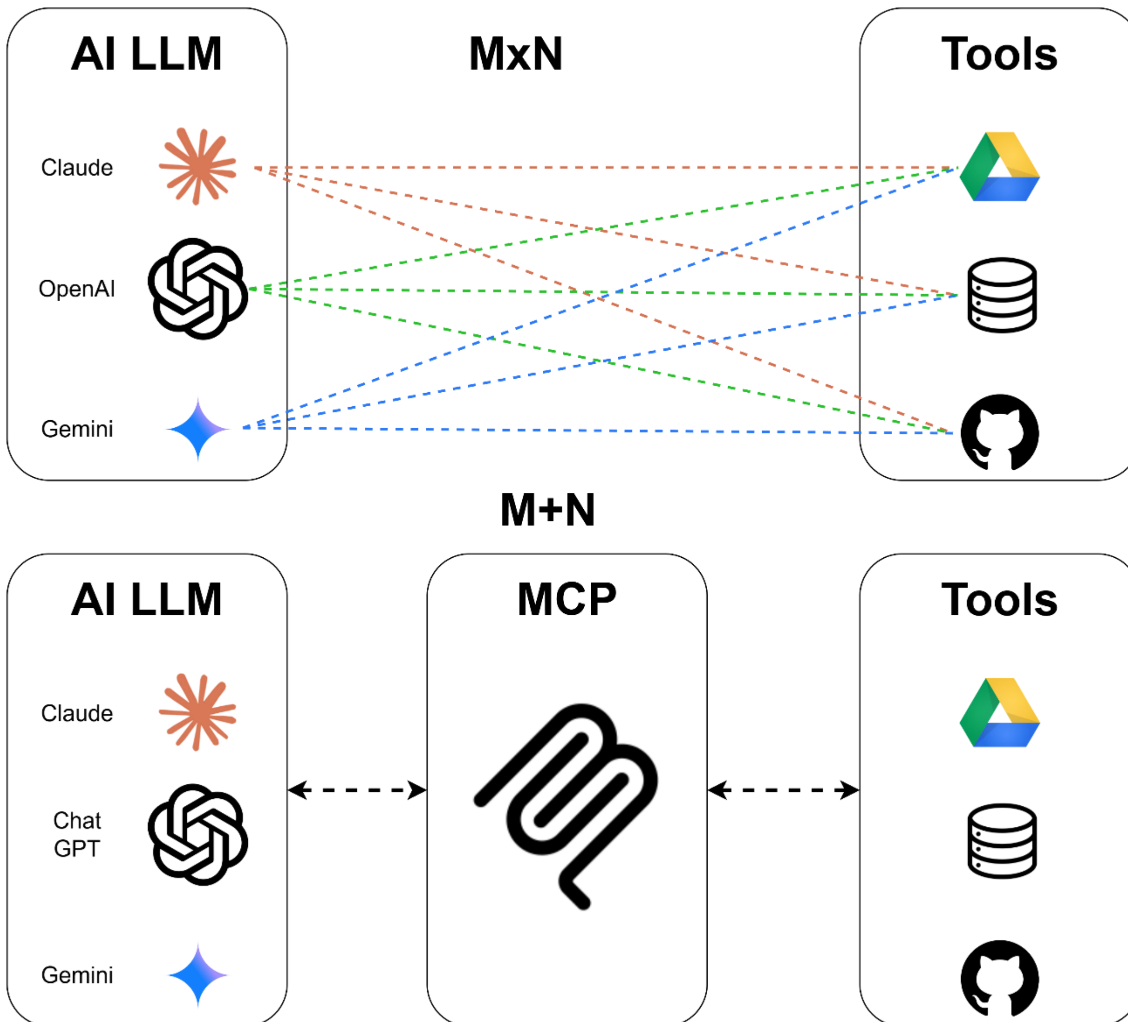


Figure 3-2 Integration of Tools with LLM without MCP and with MCP

MCP System Architecture

MCP follows a client-server pattern with four main components that work together seamlessly. Host, client, server and transport layer. Each component has its own unique function. The client and server basically send messages to each other for communication.

Hosts are LLM applications that want to access data through mcp. Hosts initiate connections. Applications like Claude Desktop, Cursor AI etc exemplify typical hosts—software that needs external data access. Hosts embed MCP clients, manage server lifecycles, and coordinate message routing. Discovery happens through JSON configuration files. Process management handles launching, monitoring, and restarting failed servers.

Clients maintain one-to-one connections with servers inside host applications. Each client represents an independent communication channel with its own state, capabilities, and message queues. State management becomes critical with multiple concurrent operations. Request queues prevent overwhelming servers—typical implementations limit concurrent requests to 50-100.

Server is a lightweight program which expose specific capabilities through standardized interfaces. Each server focuses on a particular domain—file access, database queries, API integrations, or computational tools. Connection pooling prevents resource exhaustion. Database servers maintain 10-20 connections, reusing them across requests.

Transport layers handle the actual message exchange between clients and servers. MCP defines two standard transports. Stdio uses standard input/output streams for local communication. HTTP with Server-Sent Events enables web-based communication.

The overall process is that when user ask something from LLM, inside the host application, the client which is also inside the host takes the messages to the corresponding mcp server. The responsibility of client is to take the message but also to the relevant server. The server process the message and then send back the response through the client to the LLM. LLM then shows the output to the user.

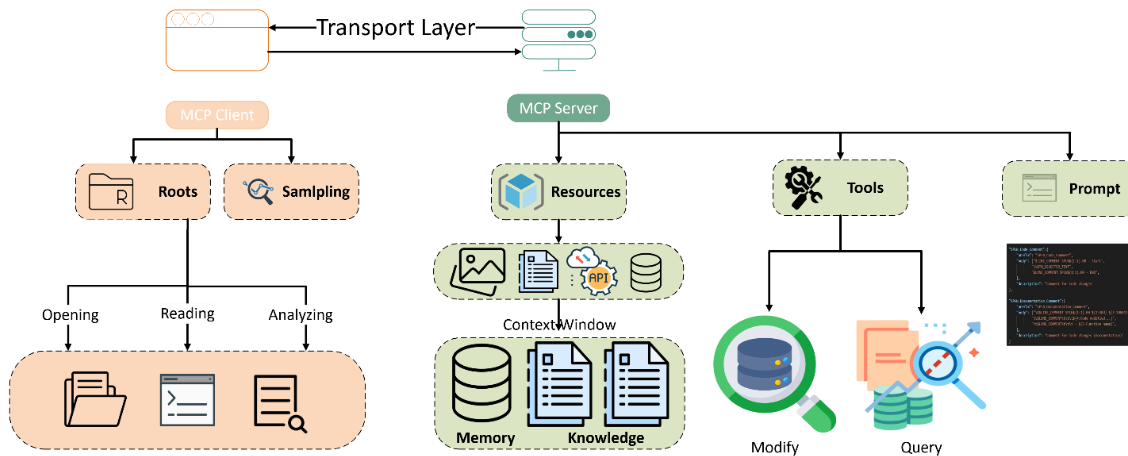


Figure 3-3 Capabilities of MCP Server

Core Concepts

Resources

Resources are similar to get requests. These are the read only data which is exposed by the server. The application can consume or use these resources but it not always brought to the context. Resources represent MCP's foundational primitive for exposing data. Each resource has a unique URI following the pattern protocol://host/path. Standard protocols include file://, postgres://, http://, and custom schemes for domain-specific resources.

Resources fall into two categories. Text resources contain UTF-8 encoded strings—source code, configuration files, logs, and structured data. Binary resources require base64 encoding—images, PDFs, audio files, and compiled binaries. Base64 encoding inflates size by 33 percent.

Discovery happens through direct listing or URI templates. Direct listing returns concrete resources with metadata. URI templates enable dynamic resource construction following RFC 6570. Templates scale better than listing thousands of individual resources.

Real-time updates work through subscriptions. Clients subscribe to specific URIs and receive notifications when content changes. File watchers detect filesystem modifications. Database triggers capture record updates.

Prompts

The prompt templates aim to achieve a very reasonable task which is the reduction in prompt engineering by the user. Prompts enable servers to define reusable templates and workflows that clients can surface to users and LLMs. Each prompt combines metadata, arguments, and message generation logic.

There are two kind of prompts one is static prompts which return fixed messages. The other is dynamic prompts, which adapt based on arguments and context. Workflow prompts guide multi-step interactions. Arguments make prompts reusable across contexts with schema validation ensuring correct usage.

Prompts can incorporate real-time data through resource embedding. Multi-step workflows build on previous responses, enabling complex guided interactions. The system supports rich interaction patterns beyond simple question-answer exchanges.

Tools

Tools represent MCP's mechanism for enabling LLMs to perform actions. Unlike resources (read-only) and prompts (templates), tools execute operations that can modify state or interact with external systems.

Tools are functions that can be invoked by the client. Each tool defines a name, description, and input schema using JSON Schema. Annotations guide AI behavior—read-only tools don't modify state, destructive tools require extra caution, idempotent tools can be safely retried.

Common tool patterns include system operations (file manipulation, process control), data processing (analysis, transformation), and external integrations (API calls, database queries). Tool composition enables complex workflows by combining multiple tools.

Sampling

Sampling reverses the typical flow, allowing servers to request LLM completions through the client. This enables sophisticated agentic behaviors while maintaining security through human oversight.

The human-in-the-loop design ensures users maintain control. Clients review requests before sending to LLMs. Users can modify prompts, reject requests, or approve with conditions.

Model selection balances multiple factors. Hints suggest preferred models. Priority values weight cost, speed, and intelligence. The client makes final selection based on available models and preferences.

Common patterns include clarification requests when user intent is ambiguous, multi-step reasoning for complex tasks, and dynamic documentation generation based on context.

Roots

Roots define operational boundaries for MCP servers. Clients declare which roots servers should focus on—URIs that suggest relevant resources and locations.

Common use cases include project directories, repository locations, API endpoints, and configuration paths. While primarily used for filesystem paths, roots can be any valid URI.

Servers interpret roots based on their capabilities. File servers use roots to filter accessible paths. Database servers might limit queries to specific schemas. API servers could restrict endpoints.

Root-based security isolation enforces boundaries. Resources outside declared roots may be inaccessible. Cross-root operations might require additional permissions. Each root can have independent access policies

Transports

Transports provide the communication foundation for MCP. All transports implement a common interface enabling diverse mechanisms while maintaining consistency.

Stdio transport leverages standard input/output streams. Perfect for local processes, it offers minimal latency—typically under 5ms for round trips. Messages flow through newline-delimited JSON. Buffer management handles messages up to 64KB by default.

HTTP transport enables remote communication. Clients POST JSON-RPC messages to server endpoints. Servers respond with single JSON responses or Server-Sent Events for streaming. Authentication happens through standard HTTP headers.

Performance varies significantly. Stdio achieves sub-millisecond latency for small messages. HTTP adds 10-50ms depending on network conditions. Large messages over 1MB stress both transports differently.

4 CO PILOTS FOR BIM SOFTWARES

4.1 BIMCP

The BIMCP means “Building Information Model Context Protocol written in python language”. Its name shows the idea that this project is the integration of BIM and AI through model context protocol. The system was intended to integrates multiple BIM softwares to address fundamental challenges in building information modeling workflows but for now it is only developed for Autodesk Revit. Natural language interaction with BIM systems becomes possible through this multi-layer architecture. The implementation emerged through multiple techniques applied to use mcp server in over 3 months to get this fully functional project.

4.1.1 System Architecture Overview

The main components of this system are MCP and Revit application and the socket communication between these two. Each one has sub components. For MCP, the architecture is discussed in the previous chapter which contains client inside host application and mcp server. The host application also contain LLM which in this case is the claude LLM because Claude Desktop was used for connecting mcp with Revit. The application part comprise of Revit Plugin and Revit software itself which contains the revit plugin. Each of these are explained in the next section.

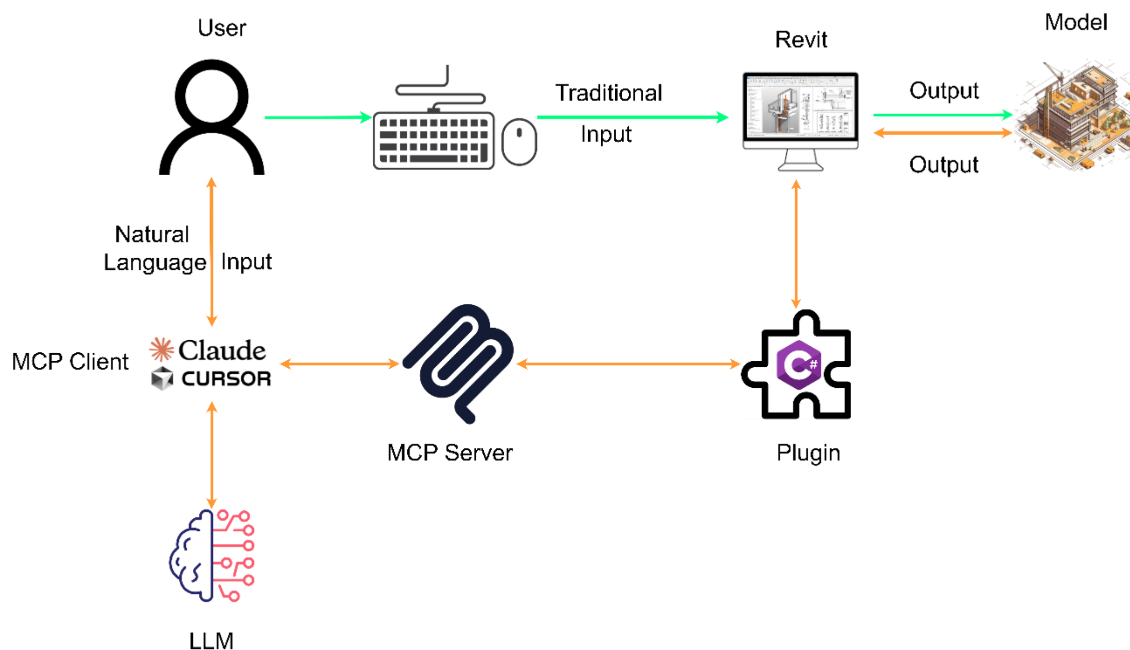


Figure 4-1 Tradional and Natural language wrokflow in Revit

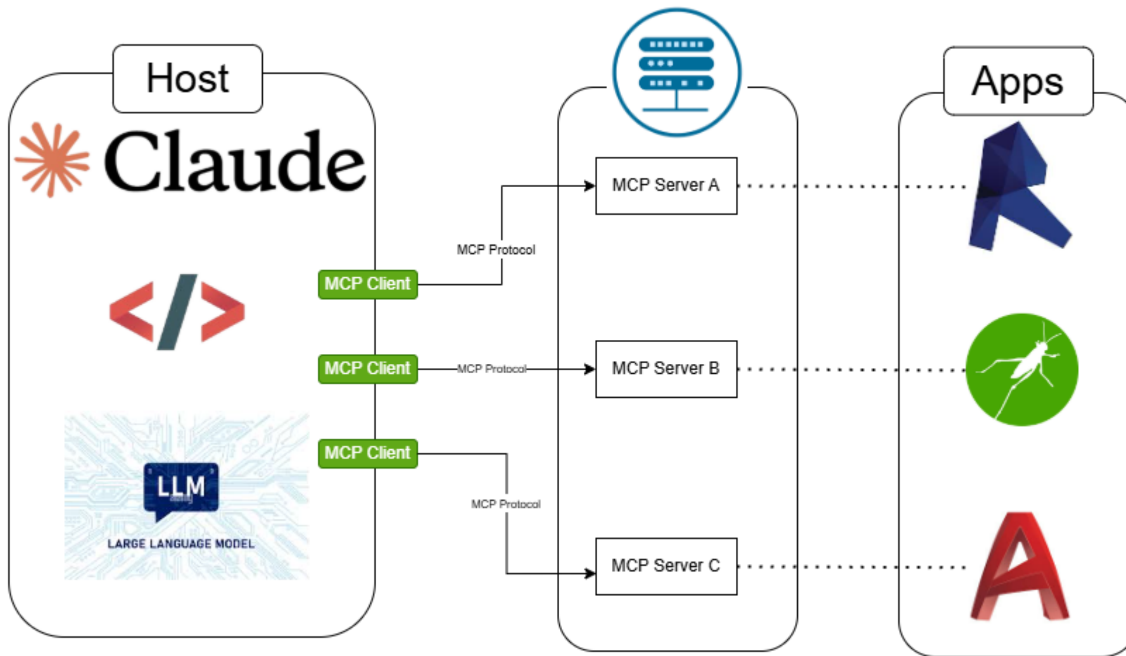


Figure 4-2 Integration of softwares with MCP

MCP Server

The mcp server contains Resources, Prompts and Tool. These three components work together to perform action on Revit model either creating geometry or querying information. Each component is registered in its corresponding separate file. For example for tool.py, resourses.py and prompts.py. Then these file are collectively registered in main server.py file. All of the three can be registered at the same time in main server.py file but its confusing and error in one part of the code can disturbed the whole functionality of the server so its better to have separate files even if one file or a part of file has error it can be easily inspected and debugged in the mcp inspector and can be corrected. It is explained in the mcp inspector and debugging section in detail.

```
from mcp.server.fastmcp import FastMCP
mcp_server = FastMCP(Python-BIMCP)

def run_server(transport: str = "stdio", port: int = 8001) -> None:
    register_all_tool(mcp_server)
    register_all_resources(mcp_server)
    register_prompts(mcp_server)
```

Tools

In this server there is only one tool which is the execute_revit_code which is used for sending C# code to Revit plugin which compiles the code and run it in Revit as API. Implementation in server.py uses FastMCP from the official Python MCP SDK for protocol handling. The code is generated by the LLM and then the tool function is only used for sending the code to Revit plugin which means that the tool is not generating a C# code but it is the LLM which is responsible for this.

Many tools can be added to this server for example get_selected, which can be used for getting information about selected elements in revit and many more tools like this kind. These tools will bypass the execute_revit_code tool which is used for writing a fresh code each time it is called. But for this project it is kept simple to just have idea how LLM can generate perfect code for each operation with the help of mcp.

```
@mcp_server.tool()
async def execute_revit_code(
    code: str,
    use_transaction: bool = False,
    transaction_name: str = "API Operation",
    transaction_strategy: str = "single",
    ctx: Context = None
) -> Dict[str, Any]:
```

The @mcp_server.tool() decorator acts like a smart assistant that handles the messy details. Before running anything, it ensures all parameters are correct (right types, required fields present). It converts MCP protocol messages into regular Python function calls - developers don't need to worry about protocol details. If something goes wrong, it packages the error in a standard format that Claude can understand. For operations that take a long time, it can update Claude with progress messages like "Step 2 of 5..."

The tool is developed with a built-in "instruction manual" for Revit. The tool's documentation tells AI assistants that Revit uses old C# (version 7.3), so fancy new features won't work. It explains when code needs to be wrapped in transactions (like when changing the model). It reminds users that Revit thinks in feet, not meters - so conversions are needed: millimeters ÷ 304.8 = feet.

These instructions are also added as a prompt which is presented in the prompts section.

Resources

The resource implementation represents one of the most extensive components of the BIMCP system. Multiple distinct BIM operations are exposed through a standardized URI-based interface. This resource system transforms Revit models into queryable data sources accessible via MCP and these data sources can be bring in to context either by the client itself or by the user to update the LLM for next operation.

As it is discussed in the previous section, that resources are registered in a separate file but inside that file each resource is implemented through a constant pattern which make it quick readable and also maintainable.

```
@mcp_server.resource("revit://model/info")
async def model_info() -> Dict[str, Any]:
    """Get information about the current Revit model."""
    logger.info("Resource accessed: revit://model/info")
    return await get_revit_model_info()
```

Resources create C# code fresh each time you ask for data, instead of storing old information. When they send data back, they include everything you might need - summaries, details about types, and useful numbers. If something goes wrong with the connection, they don't crash - they send back helpful error messages instead. All resources work in the background without blocking other operations, so the server stays fast and responsive even when handling multiple requests.

The resource system works in a hierarchical URI schema which provides intuitive navigation while maintaining extensibility for future additions. The revit:// protocol clearly indicates domain-specific resources preventing confusion with web protocols.

```
# Resource URI Structure
revit://category/subcategory/specific-resource

# Actually implemented resources in your project:
revit://model/info                                  # General model information
revit://model/health                                # Model health and diagnostics
revit://elements/walls                              # Wall elements
revit://elements/columns                            # Structural columns
revit://elements/beams                              # Structural beams
revit://elements/floors                             # Floor elements
revit://elements/levels                             # Building levels
revit://elements/rooms                              # Room spaces
revit://elements/openings                           # Doors and windows combined
revit://elements/views                              # All view types
revit://validation/warnings                         # Model warnings
revit://structural/analysis                         # Structural analysis
revit://structural/connections                      # Connection details
revit://structural/reinforcement                    # Rebar information
revit://schedules/quantities                        # Quantity takeoffs
revit://schedules/cost-estimation                   # Cost calculations
revit://schedules/specifications                    # Material specifications
revit://schedules/custom                            # Custom schedules
revit://geometry/materials                          # Material geometry
revit://geometry/spatial                            # Spatial relationships
revit://validation/standards-compliance/
        slovenian-residential                       # Standards compliance
```

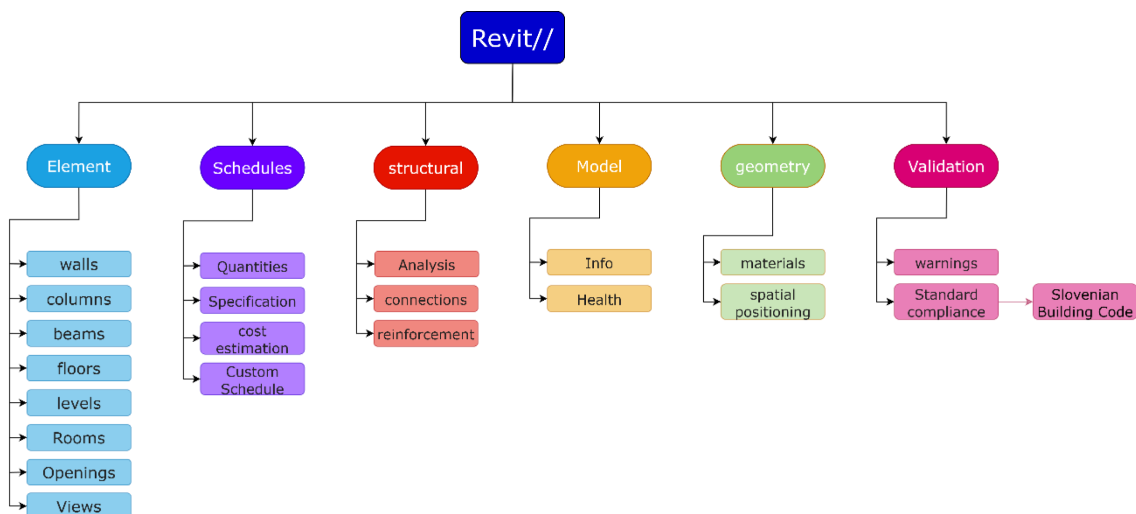


Figure 4-3 Resources developed in BIMCP

One of the example resource's code snippets for `("revit://model/info")` is given below:

```
{
  "fileName": "ASN-STR-ZONE2-MA-SDR_R00",
  "pathName": "D:\\REVIT\\Structural\\01-Structure\\ASN-STR-ZONE2-MA-SDR_R00.rvt",
  "isModified": false,
  "version": "Autodesk Revit 2024",
  "build": "24.3.10.22",
  "totalElements": 15755,
  "viewCount": 102,
  "sheetCount": 13,
  "categories": {
    "Phases": 2,
    "Materials": 106,
    "Levels": 8,
    "Primary Contours": 2,
    "Area Schemes": 2,
    "Elevations": 6,
    "Views": 75,
    "Shared Site": 2,
    "Work Plane Grid": 42,
    "Revision": 1,
    "Project Information": 1,
    "Structural Load Cases": 8,
    "Sun Path": 113,
    "Cameras": 5,
    "Section Boxes": 5,
    "Internal Origin": 1,
    "Color Fill Schema": 8,
    "HVAC Zones": 1,
    "HVAC Load Schedules": 25,
    "Building Type Settings": 33,
    "Space Type Settings": 125,
    "Survey Point": 1,
    "Project Base Point": 1,
    "Electrical Demand Factor Definitions": 4,
    "Electrical Load Classifications": 6,
    "Panel Schedule Templates - Branch Panel": 3,
    "Panel Schedule Templates - Data Panel": 1,
    "Panel Schedule Templates - Switchboard": 1,
    "Electrical Load Classification Parameter Element": 36,
    "Material Assets": 66,
```

```
    "Pipe Segments": 13,
    "Legend Components": 105,
    "Grids": 12,
    "Floors": 40,
    "Span Direction Symbol": 113,
    "<Sketch>": 305,
    "Dimensions": 256,
    "Structural Foundations": 29,
    "Walls": 20,
    "Structural Columns": 123,
    "Structural Framing": 524,
    "Foundation Span Direction Symbol": 5,
    "Spot Elevations": 33,
    "Automatic Sketch Dimensions": 64,
    "Shaft Openings": 3,
    "Stairs": 4,
    "Runs": 8,
    "Landings": 4,
    "Stair Paths": 9,
    "Railings": 8,
    "Top Rails": 8,
    "Reference Planes": 36,
    "Railing Rail Path Extension Lines": 40,
    "Balusters": 40,
    "Structural Framing Tags": 186,
    "Structural Rebar": 4697,
    "Structural Area Reinforcement": 5,
    "Structural Area Reinforcement Symbols": 10,
    "Structural Area Reinforcement Tags": 5,
    "Constraints": 15,
    "Windows": 10,
    "Scope Boxes": 2,
    "Sheets": 13,
    "Title Blocks": 13,
    "Viewports": 24,
    "Lines": 17,
    "Text Notes": 14,
    "Guide Grid": 1,
    "Matchline": 1,
    "View Reference": 4,
    "Structural Column Tags": 35,
    "Structural Foundation Tags": 37,
    "Wall Tags": 3,
    "Detail Items": 11,
    "Stair Tread/Riser Numbers": 4,
    "Structural Rebar Tags": 14,
    "Schedules": 28,
    "Revision Numbering Sequences": 2,
    "Analytical Panels": 57,
    "Analytical Members": 647,
    "Analytical Openings": 19,
    "Analytical Nodes": 2862
  }
}
```

Prompts:

The third important component of mcp server is the prompt which isd discussed in previous chapter in detail. This Python_BIMCP server contain three prompts i-e revit_coding_standards, model_overview and analyze_walls but the important one is the only revit_coding_standards prompt. This prompt contains all the instructions and code example which are necessary for LLM to write the code in ensuring that AI-generated code is compatible with Revit's .NET Framework 4.8 environment, which requires C# 7.3 syntax.

The prompt is designed in five parts.

Compatibility Rules (22 Rules)

Transaction Management

Unit Conversion Standards

Best Practices

Code Examples

Each one has an equal important role. The 22 rules which are given below enforces restrictions on the modern C# features which are not supported in the Revit;s runtime environment.

```
# COMPREHENSIVE REVIT CODING STANDARDS

## BASE COMPATIBILITY RULES

1. NO string interpolation (`$"..."`)- Use string.Format() or + concatenation instead.
2. NO null conditional operators (`?.`)- Use explicit null checks instead.
3. NO LINQ extension methods (.Where(), .Select(), etc.)- Use traditional foreach loops.
4. NO pattern matching (`if (obj is Type t)`)- Use traditional casting (`Type t = obj as Type; if (t != null)`).
5. NO expression-bodied members (`Method() => expression`)- Use traditional method bodies with return statements.
6. NO tuple returns or deconstruction- Use traditional classes or out parameters.
7. Always use explicit braces `{ }` for control structures, even for single-line blocks.
8. Set the 'result' variable to return data before using return statements.
9. Always activate family symbols before use: `if (!type.IsActive) { type.Activate(); }`.
10. Always use proper transaction management with try/catch and rollback on error.
11. Verify elements exist after creation with element.IsValidObject checks.
12. Remember Revit uses feet as internal unit (1.0 = 1 foot).
13. Use .Value instead of .IntegerValue with ElementIds in newer Revit versions.
14. NO .Cast<T>().ToList() chaining - Use Cast<T>() then ToList() as separate operations.
15. NO OrderBy().ToList() chaining - Use OrderBy() then ToList() as separate operations.
16. Always handle name conflicts when setting element names - use try/catch around Name property.
17. Use tolerance when comparing doubles (e.g., Math.Abs(a - b) < 0.001).
18. Check for existing elements before creating duplicates (levels, types, etc.).
19. Always use 304.8 for mm to feet conversion (feet = mm / 304.8).
20. Return structured dictionaries instead of simple strings for better error handling.
21. NEVER create transactions when using execute_revit_code with use_transaction=true.
22. The execute_revit_code tool auto-wraps code in a transaction when use_transaction=true.
```

The transaction management explains how to safely make changes to a Revit model. In Revit, any modification (like creating a wall or moving a door) must happen inside a transaction - a protective wrapper that allows changes to be undone if errors occur. The prompt teaches that the execute_revit_code tool automatically creates this protective wrapper, so developers don't need to add their own transaction code (which would cause errors, like trying to edit a locked element).

The unit conversion standards addresses the revit's internal unit system which is feet even if the user works in metric units. The prompt provides conversion formula (feet = millimeters / 304.8) so that when a user wants to create a 3-meter wall, the LLM knows to convert 3000 millimeters to 9.84 feet before sending the command to Revit.

The best practices part provides patterns for reliable Revit operations. These include verifying elements exist after creation (because creation can fail silently), checking that parameters have values before accessing them (to prevent crashes), activating family types before use, returning detailed success/failure information instead of simple messages, and using filtering methods to find elements in large models without performance issues.

Finally the code examples which contains over 50 working examples covering common BIM operations like creating walls, levels, grids, columns, and beams are used for creating models. These code examples give idea to LLM that how elements will be created through coding which are pretested before adding it to this prompt. The LLM uses this as template and just put the information which user gives to it and then it is send to revit for execution. All these parts of prompts work at the same time which make the LLM model align to what is needed and how to achieve it. An example for code standard is given below

````
## EXAMPLE OF IMPROVED COMPATIBLE CODE

```csharp
// Create a transaction
using (Transaction tx = new Transaction(doc, "Create Wall"))
{
    try
    {
        tx.Start();

        // Get wall type (with proper null checks)
        ElementId wallTypeId = null;
        FilteredElementCollector collector = new FilteredElementCollector(doc);
        collector.OfClass(typeof(WallType));
        WallType wallType = null;

        foreach (Element elem in collector)
        {
            wallType = elem as WallType;
            if (wallType != null)
            {
                break;
            }
        }

        if (wallType != null)
        {
            wallTypeId = wallType.Id;

            // Activate the type if needed
            if (!wallType.IsActive)
            {
                wallType.Activate();
            }

            // Create wall curve with safety check
````
````

````
            XYZ start = new XYZ(0, 0, 0);
            XYZ end = new XYZ(10, 0, 0);

            if (start.DistanceTo(end) > 0.00001)
            {
                Line line = Line.CreateBound(start, end);

                // Get level
                Level level = null;
                FilteredElementCollector levelCollector = new FilteredElementCollector(doc);
                levelCollector.OfClass(typeof(Level));

                foreach (Element elem in levelCollector)
                {
                    level = elem as Level;
                    if (level != null)
                    {
                        break;
                    }
                }

                if (level != null)
                {
                    // Create wall with proper parameters
                    Wall wall = Wall.Create(
                        doc,
                        line,
                        wallTypeId,
                        level.Id,
                        10.0,   // Height in feet
                        0.0,    // Offset
                        false,  // Flip
                        false); // Structural

                    // Verify wall was created successfully
                    if (wall != null && wall.IsValidObject)
                    {
                        // Success - create result
                        ElementCreationResult result = ElementCreationResult.CreateSuccess(
                            wall.Id,
                            string.Format("Wall created: ID {0}", wall.Id.Value));

                        tx.Commit();
                        return result;
                    }
                    else
                    {
                        tx.RollBack();
                        return RevitOperationResult.CreateFailure("Failed to create wall");
                    }
                }
                else
                {
                    tx.RollBack();
                    return RevitOperationResult.CreateFailure("No levels found in the model");
                }
            }
            else
            {
                tx.RollBack();
                return RevitOperationResult.CreateFailure("Invalid wall geometry - start and end points are too close");
            }
        }
        else
        {
            tx.RollBack();
            return RevitOperationResult.CreateFailure("No wall types found in the model");
        }
    }
    catch (Exception ex)
    {
        // Roll back the transaction on error
        if (tx.HasStarted())
        {
            tx.RollBack();
        }

        // Use exception handler for consistent error handling
        return RevitOperationResult.CreateFailure(string.Format("Error creating wall: {0}", ex.Message), ex);
    }
}
```
````
````

Tansport

This project supports two types of transport for communication which are stdio (standard input/output) and SSE (Server-Sent Event) but since the mcp server is not hosted online and is only operated locally, so the main transport is through stdio and sse is not used yet. But the configuration for web based testing is ready for future improvement.

```
from mcp.server.fastmcp import FastMCP
mcp_server = FastMCP(BIMCP)

def run_server(transport: str = "stdio", port: int = 8000) -> None:
```

Socket Communication (MCP Server-Side)

The socket-based communication framework represents the foundational transport layer enabling MCP-to-Revit interaction. The socket communication architecture implements a client-server model with deliberate role assignments. Python MCP server acts as a client connecting to the Revit plugin's embedded server. The RevitSocketClient class manages all TCP-based communication with the C# Revit plugin.

```
class RevitSocketClient:
    def __init__(
        self,
        host: str = "localhost",
        port: int = 8081,
        timeout: float = 30.0,
        max_retries: int = 3,
        retry_delay: float = 1.0
    ):
        self.host = host
        self.port = port
        self.timeout = timeout
        self.max_retries = max_retries
        self.retry_delay = retry_delay
        self.socket: Optional[socket.socket] = None
        self._connection_lock = asyncio.Lock()
```

Any application which is intended to be connected to Revit, can use port 8081 to connect. This port number can be changed if the port is used by another application, but it should be then changed in the whole system. The Revit processing time depends on the model. If it is small it takes less time otherwise more time for complex models. By default, the timeout is set to 30 sec for large operations. There is a maximum of 3 retries for Revit to attempt the code if it takes more time or fails. Revit plugins often handle multiple concurrent requests (getting elements, modifying properties, etc.) so the connection lock prevents multiple operations from using the socket connection at the same time.

JSON-RPC Message Protocol Details

The framework utilizes JSON-RPC 2.0 as the message protocol between components. This standardized approach provides structured request-response patterns with proper error handling:

```
async def send_request(
    self,
    method: str,
    params: Optional[Dict[str, Any]] = None
) -> Dict[str, Any]:
    if not params:
        params = {}

    await self.connect()

    request_id = str(uuid.uuid4())
    request = {
        "jsonrpc": "2.0",
        "method": method,
        "params": params,
        "id": request_id
    }

    request_data = json.dumps(request).encode('utf-8')
    self.socket.sendall(request_data)

    # Response reading with non-blocking socket
    response_data = b''
    self.socket.setblocking(False)

    # Read response in chunks until complete JSON is received
    while True:
        try:
            chunk = await asyncio.get_event_loop().sock_recv(self.socket, 4096)
            response_data += chunk
            response = json.loads(response_data.decode('utf-8'))
            break  # Complete JSON received
        except json.JSONDecodeError:
            continue  # Keep reading

    if "error" in response:
        raise RevitRequestError(
            f"Error from Revit server: {response['error'].get('message', 'Unknown error')}, "
            f"code: {response['error'].get('code', 'Unknown')}"
        )

    return response.get("result", {})
```

Each request receives a unique UUID enabling request-response correlation. The implementation handles non-blocking socket operations and reads responses in chunks until a complete JSON message is received. The framework supports both successful responses and structured error propagation.

Connection Pooling for Performance

Individual socket connections incur significant overhead for each establishment and teardown cycle. The framework addresses this through connection pooling implemented in connection_pool.py:

```
class RevitConnectionPool:
    def __init__(
        self,
        host: str = "localhost",
        port: int = 8081,
        max_connections: int = 5,
        max_retries: int = 3,
        retry_delay: float = 1.0,
        timeout: float = 30.0
    ):
        self.host = host
        self.port = port
        self.max_connections = max_connections
        self.max_retries = max_retries
        self.retry_delay = retry_delay
        self.timeout = timeout

        # Pool state
        self.connections: List[RevitSocketClient] = []
        self.available_connections: asyncio.Queue = asyncio.Queue(max_connections)
        self.pool_lock = asyncio.Lock()
        self.initialized = False
```

The pool maintains a queue of available connections with automatic lifecycle management. Connections are pre-created during pool initialization up to the configured maximum limit. The pool uses an async lock for thread-safe operations. Connection pooling improves performance for multiple concurrent requests.

Revit Plugin Integration

The C# Revit plugin serves as the execution endpoint which receives JSON-RPC requests and dynamically compiling C# code within Revit's context.

Plugin Architecture

The plugin implements as a Revit External Application, maintaining persistence throughout the Revit session. Key components include:

- Socket Server: Listens on port 8081 for incoming connections
- JSON-RPC Handler: Parses requests and formats responses
- Code Compiler: Dynamically compiles and executes C# code
- Transaction Manager: Handles Revit transactions automatically

Socket Communication (Revit Plugin-Side)

The Revit plugin implements the server-side socket communication that complements the Python MCP client. While the MCP server contains the socket client (RevitSocketClient), the Revit plugin contains the socket server that listens for incoming connections.

```
public class RevitSocketServer
{
    private TcpListener tcpListener;
    private bool isRunning = false;
    private const int PORT = 8081;

    public void Start()
    {
        tcpListener = new TcpListener(IPAddress.Loopback, PORT);
        tcpListener.Start();
```

```
        isRunning = true;

        // Start accepting connections asynchronously
        Task.Run(() => AcceptConnections());
    }

    private async Task AcceptConnections()
    {
        while (isRunning)
        {
            try
            {
                TcpClient client = await tcpListener.AcceptTcpClientAsync();
                // Handle each client connection in a separate task
                Task.Run(() => HandleClient(client));
            }
            catch (Exception ex)
            {
                LogError($"Error accepting connection: {ex.Message}");
            }
        }
    }
}
```

The socket server binds to localhost on port 8081, ensuring only local connections are accepted. This design choice enhances security by preventing remote access attempts. Each incoming connection is handled in a separate task to support multiple concurrent operations.

JSON-RPC Message Processing

When the Python MCP server sends a request to Revit, the plugin's socket server needs to receive and understand it. This process is more complex than it might seem because network messages don't always arrive in one piece.

The plugin's message processor works similarly. It reads incoming data in chunks (4KB at a time) and accumulates these chunks in a StringBuilder and when each chunk arrives, it checks if the accumulated data forms a complete JSON message. This makes it possible for a single JSON-RPC request to be split across multiple network packets, especially for large code blocks.

```
private async Task HandleClient(TcpClient client)
{
    using (NetworkStream stream = client.GetStream())
    {
        byte[] buffer = new byte[4096];
        StringBuilder messageBuilder = new StringBuilder();

        while (client.Connected)
        {
            int bytesRead = await stream.ReadAsync(buffer, 0, buffer.Length);
            if (bytesRead == 0) break;

            messageBuilder.Append(Encoding.UTF8.GetString(buffer, 0, bytesRead));

            // Try to parse complete JSON messages
            string messages = messageBuilder.ToString();
```

```
            if (IsCompleteJson(messages))
            {
                var request = JsonConvert.DeserializeObject<JsonRpcRequest>(messages);
                var response = await ProcessRequest(request);

                // Send response back
                byte[] responseData = Encoding.UTF8.GetBytes(
                    JsonConvert.SerializeObject(response)
                );
                await stream.WriteAsync(responseData, 0, responseData.Length);

                messageBuilder.Clear();
            }
        }
    }
}
```

The IsCompleteJson method counts opening and closing braces to determine if the JSON is complete. Once a complete message is detected, it's deserialized into a JsonRpcRequest object, processed, and a response is sent back through the same connection and hence it forms the biderctional communication system.

Request Routing and Execution

After receiving a complete JSON-RPC message, the plugin needs to understand what action to take. Currently, the plugin primarily handles one type of request: "execute_revit_code".

The routing system extracts the method name from the JSON-RPC request and uses a switch statement to direct it to the appropriate handler. For the execute_revit_code method, it extracts three key parameters:

- The C# code to execute
- Whether to use a transaction (for safety when modifying the model)
- The transaction name (for Revit's undo/redo system)

```
private async Task<JsonRpcResponse> ProcessRequest(JsonRpcRequest request)
{
    switch (request.Method)
    {
        case "execute_revit_code":
            // Extract parameters from the request
            var parameters = request.Params as JObject;
            string code = parameters["code"].ToString();
            bool useTransaction = parameters["useTransaction"]?.ToObject<bool>() ?? false;

            // Execute code in Revit context
            var result = await Task.Run(() =>
                RevitCodeExecutor.ExecuteCode(code, useTransaction)
            );

            // Return success response
            return new JsonRpcResponse { Id = request.Id, Result = result };
    }
}
```

The beauty of this design is its extensibility. While currently focused on code execution, the switch statement can easily accommodate new methods in the future, such as "get_element_by_id" or "create_schedule", without modifying the core communication infrastructure but for that on the server side the same tool must also be developed.

Dynamic Code Compilation

The heart of the plugin is its ability to take C# code as a string and execute it within Revit's environment. The process involves several steps that happen behind the scenes.

First, the plugin wraps the LLM generated code in a complete C# class structure. This is necessary because C# requires proper class and method definitions for compilation. The plugin automatically adds all necessary using statements for Revit API access.

```
public class RevitCodeExecutor
{
    public static object ExecuteCode(UIApplication uiApp, string code,
                                    bool useTransaction, string transactionName)
    {
        // Get Revit context objects
        UIDocument uidoc = uiApp.ActiveUIDocument;
        Document doc = uidoc.Document;
        object result = null;

        if (useTransaction)
        {
            // Wrap execution in a transaction for safety
            using (Transaction tx = new Transaction(doc, transactionName))
            {
                tx.Start();
                CompileAndExecute(code, doc, uidoc, uiApp, ref result);
                tx.Commit();
            }
        }
        else
        {
            // Execute without transaction (for read-only operations)
            CompileAndExecute(code, doc, uidoc, uiApp, ref result);
        }

        return result;
    }
}
```

The compilation process uses the CSharpCodeProvider to transform the code string into executable machine code. This happens in memory without creating physical files, making it fast and secure. The compiler is configured with references to the following assemblies:

- System.dll for basic .NET functionality
- RevitAPI.dll for Revit element manipulation
- RevitAPIUI.dll for user interface operations

If compilation fails, the plugin captures all error messages with line numbers, making it easier for LLM to debug its code. These detailed error messages are sent back to the LLM, which make corrections and retries.

Error Handling and Response Formatting

Error handling is also important when executing dynamic code in Revit. The plugin implements multiple layers of error protection to ensure that failures are informative rather than unknown.

When errors occur, they're caught at different levels:

1. Compilation errors: Syntax errors or missing references
2. Runtime errors: Null references, invalid operations
3. Transaction errors: Failed model modifications
4. Network errors: Connection issues

Each error type is formatted into a standardized JSON-RPC error response that includes:

- An error code (following JSON-RPC standards)
- A human-readable message
- Additional data like exception type and stack trace for debugging

```
public class ResponseFormatter
{
    public static string FormatError(string requestId, Exception ex)
    {
        // Create structured error response
        var response = new
        {
            jsonrpc = "2.0",
            id = requestId,
            error = new
            {
                code = -32603,  // Internal error code
                message = ex.Message,
                data = new { type = ex.GetType().Name }
            }
        };
        return JsonConvert.SerializeObject(response);
    }
}
```

This structured approach allows the AI assistant to understand what went wrong and provide helpful suggestions to users, rather than just displaying cryptic error messages.

Plugin Initialization

The plugin initializes when Revit starts, implementing the IExternalApplication interface. A message box is displayed which shows the message that revit plugin is listening to port 8081 as shown in the figure below

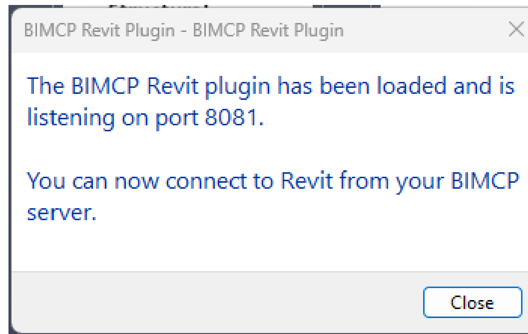


Figure 4-4 Initiliztion of Revit plugin

This architecture ensures reliable, secure, and efficient communication between the BIMCP server and the Revit environment and fully bidirectional communication occurs effectively.

4.1.2 Communication Workflow

The complete bidirectional communication flow between client (Claude in this case) and application (Revit in this case) encompasses a fourteen-step process that transforms natural language requests into BIM operations and returns structured results in the form of success or error on the basis of which LLM takes decision to end the request or try again in case of error.

Claude to Revit

When the user provides natural language request to Claude, such as "analyze all walls in the current model", the communication is initiated. Claude interprets this request and determines it requires external tool.Claude constructs a structured MCP request containing the tool identifier (execute_revit_code), the generated C# code, and relevant parameters such as transaction requirements. This request follows the MCP specification for tool invocation.

The Python MCP server receives this request through its configured transport mechanism (stdio for Claude Desktop or SSE for web clients). The FastMCP framework automatically deserializes the request and routes it to the appropriate tool handler. Parameter validation ensures all required fields are present and correctly typed.

The execute_revit_code function enhances the C# code with compatibility guidelines and necessary context. It then requests a connection from the connection pool, which manages up to five concurrent connections efficiently. The selected connection prepares the request for transmission.

The socket client formats the request using JSON-RPC 2.0 protocol, assigning a unique identifier for request-response correlation. The JSON structure includes the method name, parameters, and request ID. This message is serialized to bytes and transmitted via TCP to localhost:8081.

The C# plugin's embedded socket server receives the byte stream and reconstructs the JSON-RPC request. It extracts the C# code and compilation parameters, then dynamically compiles the code within Revit's execution environment. The compiled code executes with access to the current document and UI context.

If transactions were requested, the plugin wraps the execution in a Revit transaction for data integrity. The code runs within Revit's API context, performing the requested operations such as element creation, data extraction, or model analysis.

Figure 4-5 Claude to Revit request through natural language prompt

Revit to Claude

Upon completion, Revit stores the execution result in the designated result variable. This result contains the requested data, operation outcomes, or error information if the execution failed.

The C# plugin packages this result into a JSON-RPC response, maintaining the original request ID for correlation. The response structure includes the result data and success status, following the JSON-RPC 2.0 specification.

The socket server serializes this response to bytes and transmits it back through the established TCP connection. The data flows through the network layer in potentially multiple packets.

The Python socket client receives the incoming bytes in 4KB chunks, accumulating fragments until it can parse a complete JSON response. The client validates the response structure and correlates it with the original request using the ID.

The connection pool receives the parsed response and returns the connection to the available pool for reuse. If errors occurred, they are reformatted into structured, informative messages at this stage.

The MCP server performs final formatting based on the request type. Resources receive metadata enrichment, tools get status wrapping, and errors are transformed into user-friendly messages. The formatted response is packaged according to MCP protocol specifications.

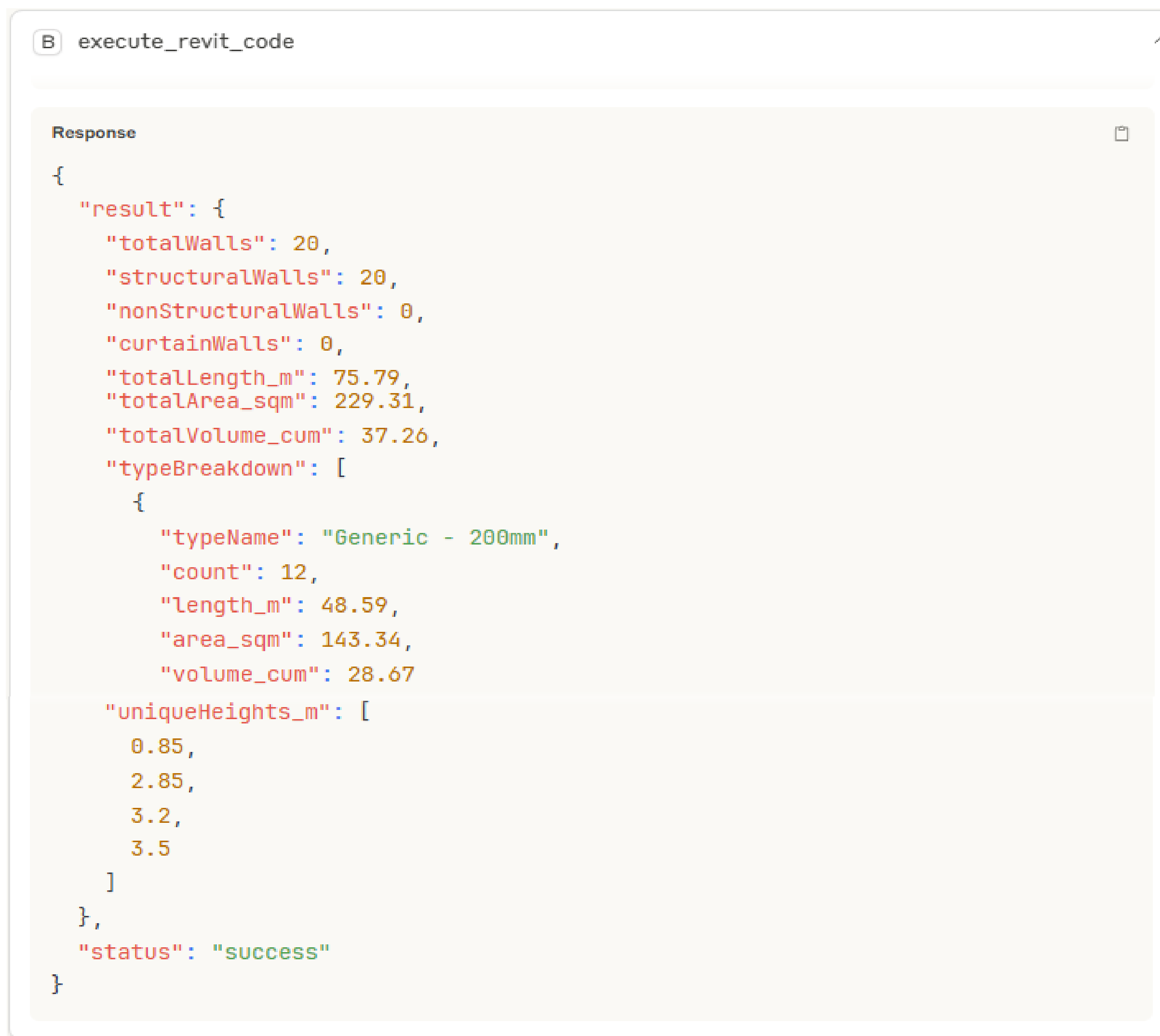


Figure 4-6 Response from Revit to Claude in JSON format

Finally, Claude receives the MCP response and interprets the technical data. Instead of displaying raw JSON, Claude generates natural language output such as "The analysis found 52 walls in the model, with 30 interior walls and 22 exterior walls. The total wall area is 1,245 square meters."

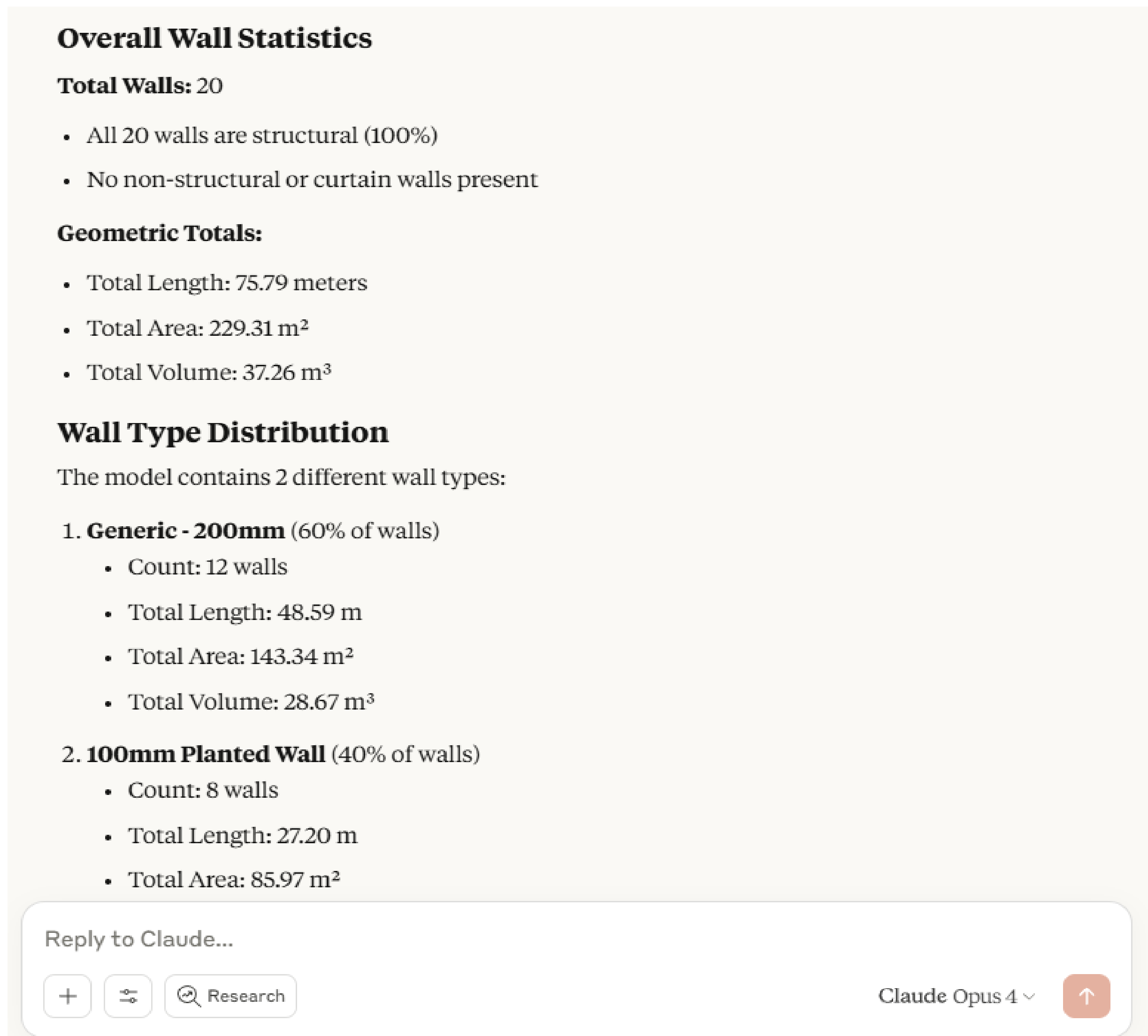


Figure 4-7 Display of information by Claude in natural language

The complete round-trip communication typically requires:

- Protocol overhead: 10-50 milliseconds
- Simple data queries: 1-2 seconds total
- Standard operations: 3-5 seconds total
- Complex model modifications: up to 30 seconds (timeout limit)

To understand each layer of this system, the complete workflow with technical details are explained in next section.

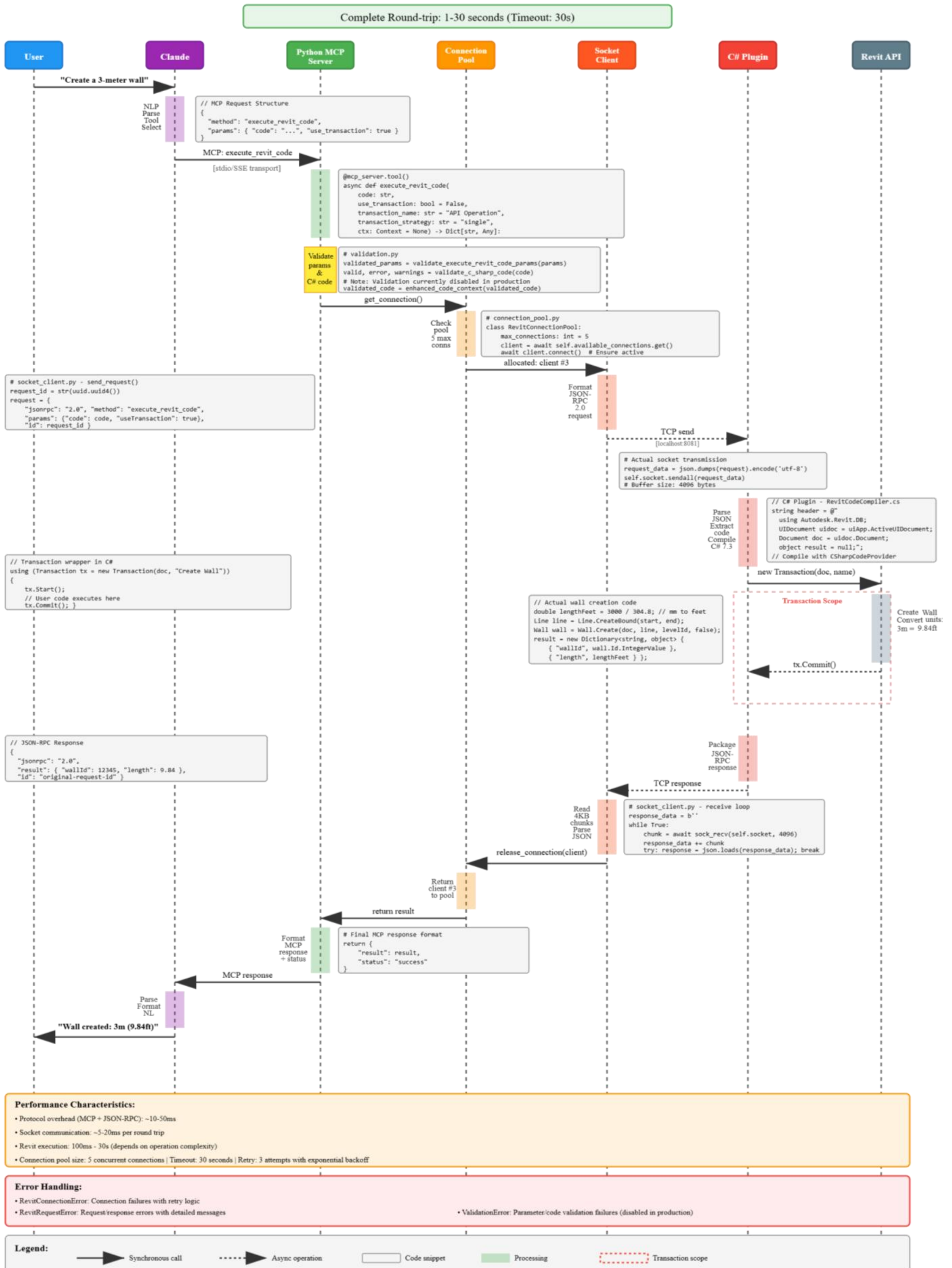


Figure 4-8 Sequence diagram of Request-Response from Claude to Revit and vice versa

4.1.3 Testing and Debugging Framework

The testing and debugging of mcp servers is not discussed in the previous chapter as it is a practical part which is explained with practical example here. The debugging of mcp server is done through mcp inspector which is a powerful tool and run with any connection with LLM specifically for resources and prompts but for tool in this project, it just show the tool list as the tool is execute_revit_code which needs an LLM written code. Therefore, the mcp inspector in this case will only focus on resources and prompts.

MCP Inspector

MCP inspector is part of debugging mcp server and is an interctive tool for testing mcp server functionality without any connection with LLM model. Here only inspection of python_BIMCP is discussed and it's the same for the BONCP also but ju

The inspector doesn't need any separate installation instead it runs through npx directly through the following code

```
npx @modelcontextprotocol/inspector
```

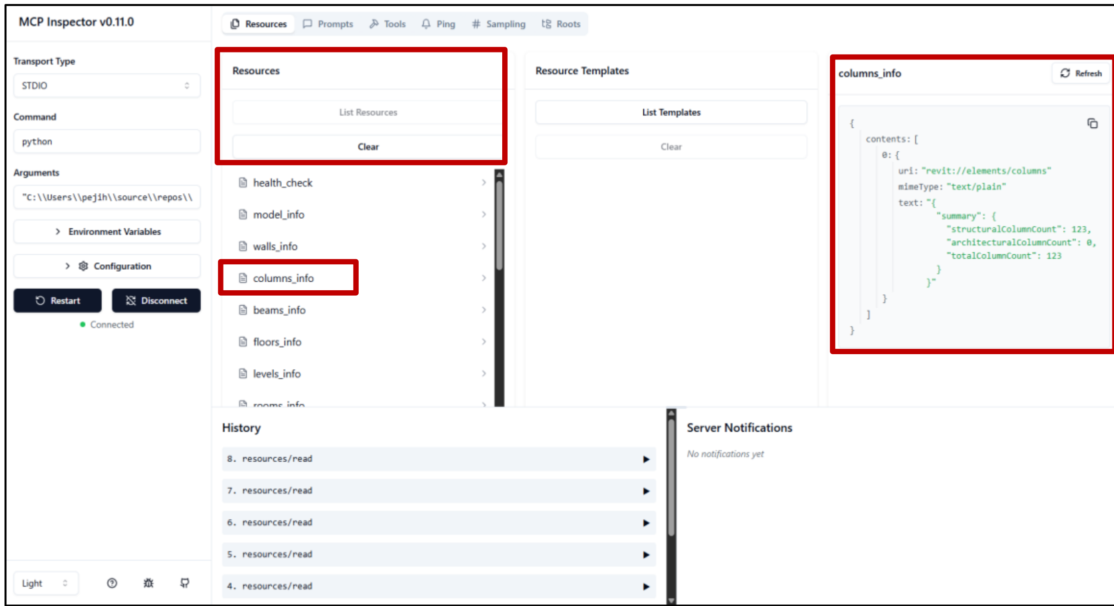
After running the code the inspector it gives a local host address in which all the tools, resources and prompts are inspected and even the connection of the mcp server is also checked here. The inspection of BIMCP server is shown in the figures below.

```
Microsoft Windows [Version 10.0.26100.4484]
(c) Microsoft Corporation. All rights reserved.

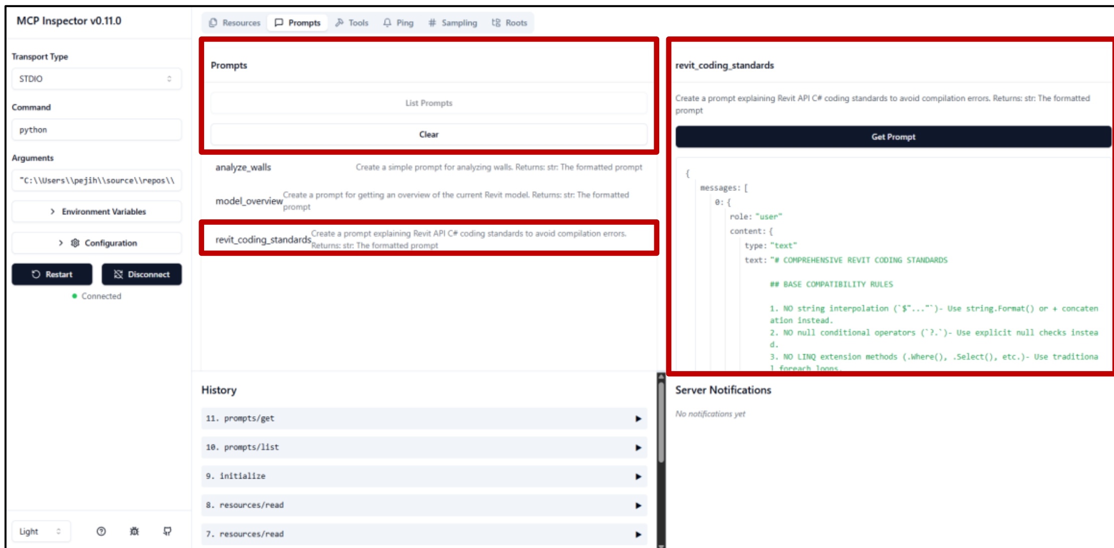
C:\Users\pejih>cd "C:\Users\pejih\source\repos\Projects-in-progress\Python-BIMCP"

C:\Users\pejih\source\repos\Projects-in-progress\Python-BIMCP>npx @modelcontextprotocol/inspector
Starting MCP inspector...
⚙️ Proxy server listening on port 6277
🔍 MCP Inspector is up and running at http://127.0.0.1:6274 🚀
```

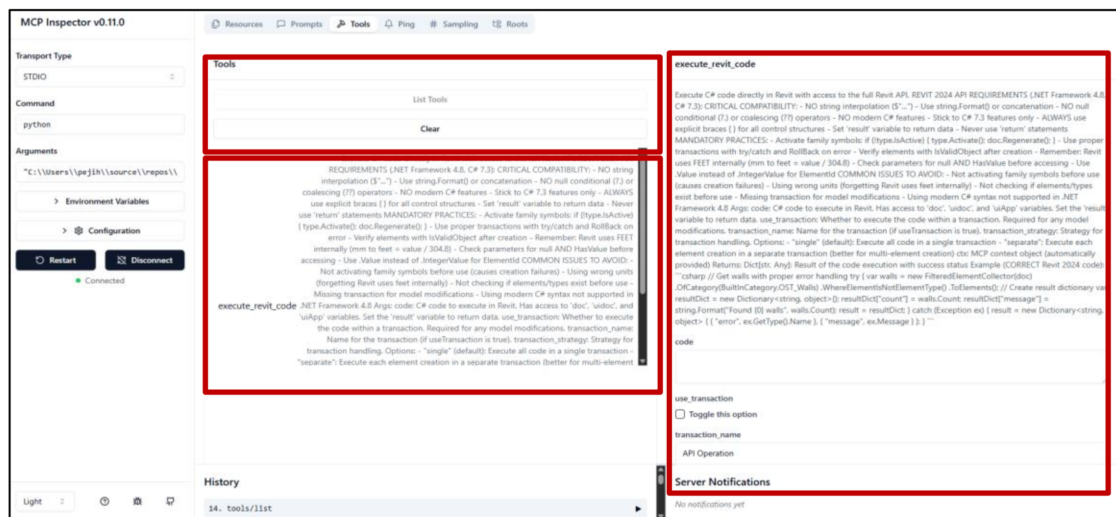
Figure 4-9 Initialization of MCP inspector



(a)



(b)



(c)

Figure 4-10 Inspection of (a) Resources (b) Prompts and (c) Tools

The first step is to select the transport type and command and argument for connecting to mcp server. In case of BIMCP it is only checked through stdio transport locally. The command and argument contain the python application and file path to the mcp server main python file respectively. All the five elements of mcp server can be inspected and debugged if there is any problem. Since this project contains only three i-e resources, prompts and tool, therefore, only these are inspected and presented here.

After the connection each tab can be checked separately. For resources after clicking the list resource button shows all the available resources added. If new resources are added and it is not visible here, it means the mcp development should be checked again so that it is added. When the resource is visible in mcp inspector then the next step is to check its functionality. It can be seen from the figure that clicking on columns_info resource open a new small window on the right showing the live information from Revit. Few columns are added in revit model and when refresh button was clicked it gave the exact number of columns. Same was for other resources.

Similarly for the prompts, the prompt tab is clicked, and it shows the list of prompts and after clicking any prompt (revit_coding_standards in this case) open a small window on the right will live prompt information. But it is not connected to Revit it is just the resuable text which can be presented to llm for instructions. It is also for testing to check whether the prompt is connected and functional or not.

In the case of tools, it is a bit different. When tool is opened, it asks for code to send it to revit and it also asks for transaction and transaction type. A manually written code was inserted initially to check if it can send the code and it worked. With this, the inspection of mcp server was performed.

After inspection the mcp server is configured either in Claude or any other client, previously discussed in chapter 3, and LLM access can be provided for testing the full functionality and same was performed with this project and it worked.

4.2 KOLEKTOR-BONCP

This is another use case of MCP and by the name it can be understood that this boncp in the connection of Bonsai with MCP. This project is developed for Kolektor Koling company which aimed to link the ifc file will LLM through blender bonsai. The need of this mcp server aroused because of large projects with a lot of elements and the increasing number of projects. This mcp server helps in getting information from ifc file and presenting in tabular format and in visual artifacts for quick readability and understanding the project. It helps in tallying information from ifc model with bill of quantities and all other documents related to the project.

This mcp server is providing a way to use intensive use of ict in construction and in future it can be extended to sensors and cameras on the site for real monitoring and data retrieving. Till now it has increased productivity of sales department and management team to take decisions and find any miss information or mistakes in boqs or ifc models which takes a lot of time and human resources to find it while LLM do it fast and accurately also. This mcp also helps the facility management for for maintenance because, various types of reports can be generated by the LLM from the ifc model for example COBie report for facility management. If the there occurs any change in the ifc file, the LLM can update the COBie report accordingly which might be time consuming when doing manually specifically if the project is large or the number of projects are more.

The Boncp is a bit different from BIMCP on the basis of what it contains. The basic development is the same, but the Boncp has more tools and no resources while the BIMCP has more resources and prompts and only one tool. The server supports extension as the basic files for resources and prompts are added but because of the tools the server is performing well and if there is a need of resources it can be added.

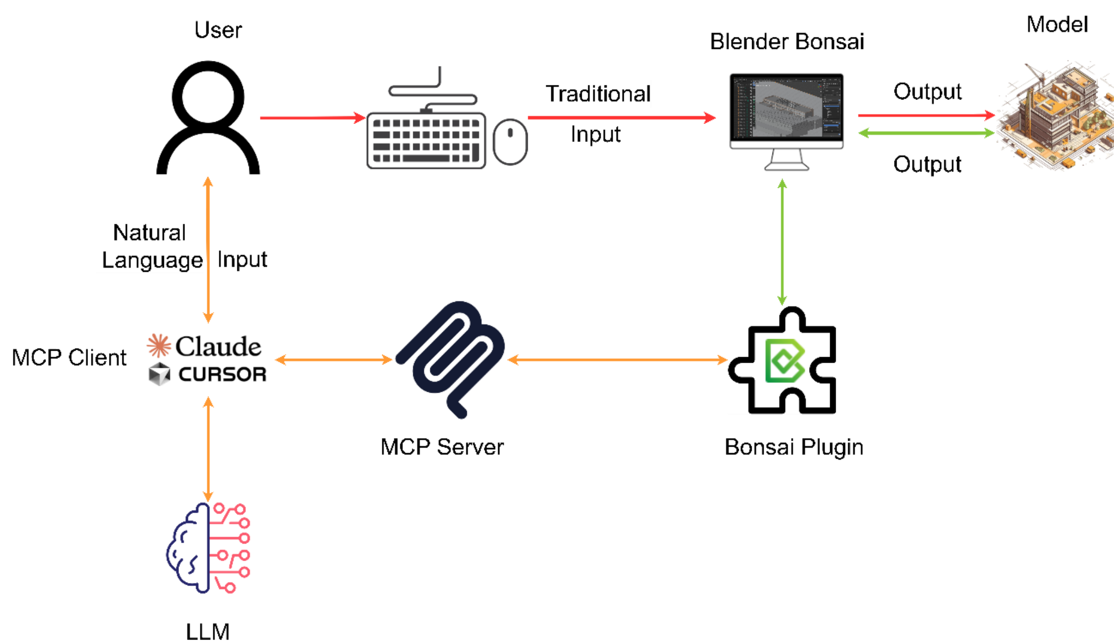


Figure 4-11Tradional and Natural language wrokflow in Bonsai

4.2.1 System Architecture

BONCP follows a similar architectural pattern to BIMCP but adapts it specifically for IFC manipulation through Blender's Bonsai addon. The system comprises two core components working in tandem. First, the Blender addon (addon.py) creates a socket server within Blender, establishing a persistent connection point at port 8002 by default which can be changed later to any port number if the port is used by other application. Second, the MCP server (tools.py) implements the Model Context Protocol using FastMCP, connecting AI assistants to Blender's IFC capabilities.

The socket server implementation in the addon demonstrates careful engineering for production reliability. Connection management handles client disconnections, automatically attempting reconnection when needed. The server runs in a separate thread to avoid blocking Blender's main UI thread—critical for maintaining responsiveness during complex IFC operations. Each client connection spawns its own handler thread, enabling concurrent request processing.

Command execution requires special handling within Blender's context. The system uses bpy.app.timers.register() to schedule command execution in the main thread, preventing threading conflicts with Blender's internal state. This approach ensures IFC modifications occur safely while maintaining system responsiveness. Error handling wraps every operation, providing detailed feedback through the socket connection when operations fail.

4.2.2 Tools

IFC-Specific Tool

The tool implementation philosophy differs fundamentally from BIMCP's approach. Rather than exposing numerous resources, BONCP implements eleven specialized tools that perform specific IFC operations. This tool-centric design was developed for the requirement of the employees of Kolektor Koling where they needed a quick access to specific information dynamically without putting themselves in the complexity of resources and prompts. The employes have to write in natural language what they need and the LLM will perform the action.

The get_ifc_project_info tool exemplifies the implementation pattern. It retrieves comprehensive project metadata including element counts by type, project descriptions, and GUID references. Entity counting covers walls, doors, windows, slabs, beams, columns, spaces, and building storeys. This provides immediate overview of project scope—critical for sales teams evaluating project complexity.

Selection-based operations represent a unique capability. The get_selected_ifc_entities and related tools work with Blender's active selection, enabling intuitive workflows. Users manually select elements in Blender's viewport, then query properties or relationships through natural language. This hybrid

approach combines visual selection precision with conversational querying power. The implementation carefully maps between Blender object IDs and IFC GlobalIds, maintaining consistency across systems.

Property extraction through get_ifc_properties demonstrates comprehensive data access. The tool leverages ifcopenshell.util.element.get_psets() to retrieve all property sets associated with entities. Results include standard IFC properties, custom property sets, and type information. This complete property access enables detailed comparisons between design intent and actual specifications—essential for quality control workflows.

4.2.3 Data Export and Analysis Capabilities

The export_ifc_data tool addresses a critical business need—transforming IFC models into analyzable datasets. Implementation supports both JSON and CSV formats, with intelligent filtering by entity type and building level. The tool flattens complex IFC hierarchies into tabular structures suitable for spreadsheet analysis. Property sets expand into individual columns, making data immediately accessible to non-technical users.

Export operations handle platform differences intelligently. The system searches for appropriate output directories based on operating system conventions—Public Documents on Windows, /tmp on Unix systems. This cross-platform consideration ensures reliable operation across diverse deployment environments. File naming includes timestamps and filter criteria, preventing accidental overwrites while maintaining traceability.

Data extraction includes comprehensive metadata. Each exported record contains ExpressId, GlobalId, IFC class, name, description, and level assignment. Type information links instances to their definitions. Property sets flatten into the export structure, creating denormalized records optimized for analysis. This approach trades storage efficiency for query simplicity—appropriate for business intelligence applications.

The extraction of GUID with elements detail to an excel sheet is important for company because all the projects are coordinated through Dalux software and in Dalux, the documents are attached to ifc elements through GUID. If an element is deleted so the link between GUID and the document is lost even if element is restored still the GUID is lost. Having a backup of these GUIDs can help in finding the appropriate docs and can be assigned to new GUIDs. While extracting GUID manually takes hours because of complex projects with a lot of elements, it is now possible with just one natural language prompt, making the workflow better and productive.

4.2.4 Visual Feedback and Model Manipulation

The get_user_view tool represents significant innovation in AI-BIM interaction. Rather than relying purely on textual descriptions, the system captures Blender's viewport as base64-encoded images. This visual feedback enables AI assistants to understand spatial context, element arrangements, and design intent that text alone cannot convey. The implementation uses Blender's screenshot operator with careful region management to capture exactly what users see.

Image compression optimization balances quality with transmission efficiency. The Python MCP server implements PIL-based compression, resizing images exceeding 800 pixels while maintaining aspect ratios. JPEG compression with quality factor 85 reduces bandwidth requirements without sacrificing visual clarity. This optimization proves essential for responsive interaction, particularly when multiple viewport captures occur during analysis sessions.

The following is the code snippet for best demonstration of this fantastic tool.

```
@mcp.tool()
def get_user_view() -> Image:
    """Capture and return the current Blender viewport as an image."""
    max_dimension = 800
    compression_quality = 85

    # Use PIL to compress the image
    from PIL import Image as PILImage
    import io
    try:
        # Get viewport image from Blender (returns base64 encoded)
        blender = get_blender_connection()
        result = blender.send_command("get_current_view")
        # Decode the base64 image data
        image_data = base64.b64decode(result["data"])
        original_width = result["width"]
        original_height = result["height"]

        # Compression is only needed if the image is large
        if original_width > 800 or original_height > 800 or len(image_data) > 1000000:
            # Open image from binary data
            img = PILImage.open(io.BytesIO(image_data))
            # Resize if needed
            if original_width > max_dimension or original_height > max_dimension:
                # Calculate new dimensions maintaining aspect ratio
                if original_width > original_height:
                    new_width = max_dimension
                    new_height = int(original_height * (max_dimension / original_width))
                else:
                    new_height = max_dimension
                    new_width = int(original_width * (max_dimension / original_height))

                # Resize using high-quality resampling
                img = img.resize((new_width, new_height), PILImage.LANCZOS)
            # Convert to RGB if needed
            if img.mode == 'RGBA':
                img = img.convert('RGB')

            # Save as JPEG with compression
            output = io.BytesIO()
            img.save(output, format='JPEG', quality=compression_quality, optimize=True)
            compressed_data = output.getvalue()
            # Return compressed image
            return Image(data=compressed_data, format="jpeg")
        else:
            # Image is small enough, return as-is
            return Image(data=image_data, format=original_format)
```

The place_ifc_object tool enables model modification through natural language. Implementation handles complex type resolution, searching the IFC model for matching element types by name. Coordinate transformation ensures correct positioning, while rotation parameters enable precise orientation control. The tool activates inactive family types automatically—a common stumbling block in manual workflows. Transaction management through Bonsai's API ensures model consistency during modifications.

4.2.5 Sequential Thinking Integration

Unique among MCP implementations, BONCP integrates the Sequential Thinking tool from the MCP servers repository. This addition transforms simple query-response interactions into structured analytical processes. The tool maintains thought history, supports branching analysis paths, and enables revision of previous conclusions. For complex IFC analysis tasks, this structured thinking approach proves invaluable.

Implementation maintains complete thought state across interactions. Each thought receives unique numbering with metadata tracking revisions and branches. The formatted output uses Unicode box-drawing characters creating visual structure in terminal environments. This presentation clarity helps users follow complex reasoning chains during multi-step analyses.

Practical applications demonstrate the tool's value. When analyzing discrepancies between IFC models and bills of quantities, the AI can systematically work through comparisons, track findings, and revise conclusions as new information emerges. The ability to branch analysis paths enables exploring alternative interpretations without losing context. This structured approach transforms ad-hoc queries into documented analytical processes.

4.2.6 Docker Deployment and Enterprise Integration

The Docker implementation elevates BONCP from experimental tool to enterprise-ready solution. The Dockerfile uses Python 3.11-slim as base, minimizing attack surface while maintaining compatibility. Dynamic configuration through environment variables enables flexible deployment across different environments. The startup script performs runtime modification of connection parameters, adapting to container networking requirements.

Container networking addresses a critical challenge—connecting to Blender running on the host machine. The solution uses host.docker.internal as the default Blender host, enabling seamless container-to-host communication. Port mapping exposes the MCP server on port 8000, standardizing access regardless of deployment method. This containerization enables consistent deployment across development, testing, and production environments.

Integration with Open WebUI demonstrates enterprise scalability potential. The containerized MCP server exposes REST/OpenAPI endpoints, enabling integration with diverse AI platforms beyond Claude. Organizations can connect the system to their preferred AI infrastructure while maintaining consistent IFC analysis capabilities. This flexibility proves essential for organizations with existing AI investments.

4.2.7 Practical Business Impact

Deployment at Kolektor Koling demonstrates measurable productivity improvements. Sales teams now analyze complex IFC models in minutes rather than hours. The ability to quickly extract element counts, verify specifications, and compare against quotations transforms pre-bid analysis. What previously required BIM specialists becomes accessible to sales staff through conversational queries.

Management teams gain unprecedented project visibility. Natural language queries like "show me all pump rooms across levels" or "list equipment requiring maintenance access" provide instant insights. The visual feedback through viewport capture enables remote project reviews without specialized software. Executives can understand project scope and complexity through AI-generated summaries rather than technical drawings.

Quality control workflows transform through automated comparison capabilities. The system identifies discrepancies between IFC models and specifications rapidly. Missing elements, incorrect quantities, or specification mismatches surface through systematic queries. This early detection prevents costly errors from propagating through project phases. The time saved in manual checking redirects to value-adding activities.

Facility management applications extend beyond initial deployment scope. The ability to generate COBie reports through natural language commands simplifies handover documentation. Maintenance schedules extract directly from IFC properties, populating asset management systems. When models update, regenerating documentation takes minutes rather than days. This agility in documentation management reduces administrative burden significantly.

4.2.8 Comparison with BIMCP

Both implementation shows the effectiveness of mcp server connection and functionality. It also shows the adaptability of mcp server to users needs as the BIMCP is the heavy resourceful mcp server while the BONCP is purely tools focused yet both perform very well.
BIMCP, in addition to modeling, uses important resource like model info for grabbing information from the model while the sequential thinking tool of the BONCP gives power for organizing the data while extracting it for normal operation or even for COBie report formation.

In terms of technicality, the BIMCP uses the python and C# language while the BONCP uses only python language throughout the system. The BIMCP, for socket communication uses the persistent connection pooling while the BONCP creates connection per request.

The detail implementation details in this chapter clearly shows how model context protocol can effectively help in natural language integration of BIM Softwares. After understanding both system now it is confirmed that model context protocol can also be extended to other BIM Softwares and applications for better coordination, productivity, time saving and accuracy.

5 SWOT ANALYSIS

5.1 STRENGTHS

Both BIMCP and BONCP proved capable of assisting in design work and data extraction. The important feature here is that BIM software talk back and forth with AI instead of just one direction like older attempts. Now architects and engineers can type regular sentences to control Revit or Blender - no coding needed, no months of training required.

The research established Model Context Protocol as an effective bridge between AI systems and professional BIM software. Unlike previous attempts that relied on brittle API wrappers or plugin-specific solutions, the MCP approach creates a standardized communication layer that works across different software platforms. This protocol-based design means the same AI assistant can potentially connect to multiple BIM applications without requiring separate integrations for each one.

The technical setup works reliably, specifically the socket connection keeps data flowing smoothly between parts which enables BIMCP to juggle up to five tasks at once without breaking connection. Simple operation takes 1-2 second in execution while for complex tasks it depends but it doesn't go beyond 30 seconds as this maximum limit for timeout.

The BIMCP system includes a runtime code compiler that takes AI-generated C# code and executes it safely within Revit's environment. This required solving the technical challenge of validating modern AI-generated code against .NET Framework 4.8 limitations while providing helpful error messages when compilation fails. The system automatically wraps user code in proper transaction contexts and handles memory management, making it possible for non-programmers to execute complex BIM operations through natural language.

Kolektor Koling put BONCP to work and observed that data analysis related tasks that took hours now wrap up in minutes. Getting GUID numbers out for Dalux used to mean hours of clicking and copying while sometime impossible after many revisions - now it's one sentence typed in plain language. This isn't just theory anymore; it's helping a real company get work done faster.

BONCP does something clever with screenshots. It grabs what's on screen in Blender and lets the AI see it, not just read about it. This helps the AI understand space and position better than words alone. The Docker setup means big companies can run it properly, and since it's open source, anyone can tweak it for specific needs.

5.2 WEAKNESSES

Some big problems hold these systems back. BIMCP only runs on Windows because that's all Revit supports. Mac and Linux users are out of luck unless switching operating systems happens.

Money becomes an issue fast. Claude costs €20 monthly just for basic use. Heavy users pay more from €90 to €180 . Small firms cannot afford these high costs, which goes against the whole open-source idea of the project.

Revit still uses .NET Framework 4.8, making it necessary to write C# code in older version. Newer, better programming tricks are off limits.

The system can only be operated by single user and team collaboration and conversation with AI is not possible at the time.

The AI system, even after adding prompts in mcp server, still work inefficiently with bad prompting from the user. For example if user says "put a door in the middle of the wall" when five walls exist, and it doesn't know which one. No way to point or pick to clear things up which makes it necessary for user to learn prompt engineering.

Big projects with a lot of data makes the ai use more tokens and the subscription limits reaches quickly which compels the user to wait for the system to be restored. Similar is the case with BONCP in which the screenshot option eats a lot of token as processing image takes more power than text.

5.3 OPPORTUNITIES

Plenty of room to grow here. Switching to free AI models like DeepSeek or LLaMA would eliminate the subscription fees. Companies could teach these models building terms and rules through AI engineers, making responses more accurate for specific work.

Other BIM software needs this too. ArchiCAD, BricsCAD, Tekla, ETABS, SAP200 etc users would benefit from the same natural language control. The MCP setup already works, so adapting it for other programs shouldn't start from scratch.

Adding sketch input or hand gestures and voice command would make it more powerful. Workers on site could point at actual spots while talking, and the system would understand both. Much more natural than typing everything. This function is used by google AI Studio but integration with MCP requires much understanding.

Hooking up building sensors and performance data opens new doors. Architects could ask "How much energy do office buildings like this use here?" and get answers from real buildings with real measurements. From design through construction to maintenance, everything stays connected.

Moving to web browsers will make it easily accessible anywhere through internet with no installation, work from anywhere, any computer. Perfect for consultants who need occasional access or teams spread across offices. Cloud hosting also means better connections to other online tools.

5.4 THREATS

Things change fast in AI, and that's risky. New AI models might need complete rewrites of the system. Autodesk could update Revit in ways that break everything. Constant fixes and updates drain time and money.

Big software companies might lock out third-party tools. These companies have money to build slick alternatives that look better even if working worse. Marketing power beats technical quality sometimes.

Many architects and engineers don't trust AI for serious design work. Who's liable if the AI suggests something that fails? Current laws don't cover AI-assisted design clearly. Some companies also don’t want to expose data to AI. This uncertainty scares off potential users.

Depending on other companies' products creates problems. Revit, Blender, or Claude could change terms, raise prices, or shut down. Any of those kills the system. Some projects involve secret or sensitive buildings where using cloud AI raises security red flags.

If people think AI will steal jobs rather than help work better, resistance will grow. Professional groups might push for rules blocking AI design tools, especially if high-profile failures happen.

5.5 STRATEGIC ASSESSMENT

Looking at everything together, these systems have strong foundations. The good outweighs the bad, and most problems come from current software limits, not bad design choices. Kolektor Koling's success proves the concept works in real business.

The construction industry needs better digital tools and higher productivity - these systems deliver both. Being open source and protocol-based means growing and changing with technology instead of becoming obsolete remains possible. Next steps should focus on breaking free from Windows-only limits, cutting subscription costs, and connecting to more BIM software.

This research built a foundation others can build on to keep improving how humans and computers work together in technical fields.

6 CONCLUSION

6.1 SUMMARY OF RESEARCH ACHIEVEMENTS

This research successfully demonstrates the integration of Large Language Models with Building Information Modeling environments through the Model Context Protocol (MCP). The main achievement establishes two-way communication between AI assistants and professional BIM software, changing the traditional one-way command-response pattern into genuine conversation.

Two complete implementations validate the research approach: BIMCP (Building Information Model Context Protocol) for Autodesk Revit integration and Kolektor-BONCP for IFC manipulation through Blender's Bonsai addon. These systems provide practical solutions to long-standing challenges in BIM accessibility, showing that natural language interfaces can bridge the gap between human design intent and complex software operations.

The research establishes MCP as a standard communication protocol that supports different BIM applications without requiring software-specific integrations. This protocol-based approach represents a significant advancement over previous attempts that relied on fragile API wrappers or custom plugin solutions. The standardization enables consistent AI interaction patterns across different BIM platforms.

The practical validation at Kolektor Koling provides concrete evidence of measurable productivity improvements. Tasks that previously required hours of manual work, such as extracting GUID information for Dalux coordination or generating compliance reports, now complete in minutes through conversational queries. This real-world implementation demonstrates the transition from experimental technology to production-ready business solution.

6.2 ADDRESSING THE RESEARCH QUESTIONS

6.2.1 Primary Research Question

The primary research question—"How can Large Language models be integrated with Building Information Modeling environments to enable direct, bidirectional interaction that assists architects, engineers and construction industry professionals?"—has been answered through the development and deployment of functional MCP-based systems.

The integration works through a multi-layered architecture comprising four essential components: the AI assistant (Claude), the MCP server implementation, socket-based communication protocols, and embedded plugins within target BIM software. This architecture enables genuine two-way communication where the BIM system can request clarification, report operation status, and provide contextual feedback.

6.2.2 Subsidiary Research Questions

RQ1 concerning technical architectures for reliable communication has been addressed through the implementation of robust socket-based communication with connection pooling, JSON-RPC 2.0 protocol compliance, and comprehensive error handling mechanisms. The BIMCP system demonstrates sustained performance with up to five concurrent connections and sub-30-second response times for complex operations.

RQ2 regarding the effectiveness of natural language commands versus traditional programming approaches reveals significant advantages in accessibility and learning curve reduction. Natural language interfaces eliminate the need for specialized programming knowledge, enabling architects and engineers to directly control BIM software through conversational interaction. However, semantic challenges persist in translating ambiguous design intent into precise BIM operations, particularly when spatial references lack explicit context.

RQ3 exploring technical constraints within existing software ecosystems identifies several critical limitations and corresponding solutions. Platform dependencies, particularly Revit's Windows-only operation and .NET Framework 4.8 constraints, require careful architectural decisions and compatibility layers. The research demonstrates that these constraints can be effectively managed through runtime code compilation, transaction management systems, and comprehensive validation frameworks.

6.3 CONTRIBUTIONS TO AEC INDUSTRY

This research contributes significantly to the Architecture, Engineering, and Construction (AEC) industry's digital transformation by making advanced BIM capabilities accessible to more users. The natural language interface removes traditional barriers that prevented non-technical stakeholders from directly interacting with complex BIM data, enabling broader participation in digital workflows across project teams.

The standardization of AI-BIM communication through MCP establishes a foundation for industry-wide adoption of intelligent design assistance. Unlike proprietary solutions that lock organizations into specific software ecosystems, the open-source MCP approach enables interoperability and reduces vendor dependency.

The practical deployment at Kolektor Koling demonstrates measurable business value through dramatic productivity improvements in data analysis, quality control, and project coordination tasks. The transformation of multi-hour manual processes into minute-long conversational queries represents a paradigm shift in how construction professionals interact with project data.

The research establishes a new category of "BIM Copilots" that extend beyond simple automation to provide intelligent assistance throughout the building lifecycle. From initial design through construction coordination to facility management, these AI assistants enable continuous engagement with building data. The visual feedback capabilities demonstrated in BONCP particularly enhance remote collaboration and quality assurance processes.

6.4 TECHNICAL ACHIEVEMENTS AND INNOVATION

The technical achievements encompass several solutions to complex integration challenges. The dynamic C# code compilation system in BIMCP represents a significant advancement in runtime code generation for BIM applications. This system safely executes AI-generated code within Revit's protected environment while maintaining model integrity through comprehensive transaction management and error handling.

The socket-based communication architecture implements sophisticated connection pooling and message queuing mechanisms that ensure reliable performance under concurrent load. The JSON-RPC 2.0 protocol implementation provides standardized request-response patterns with comprehensive error propagation, enabling detailed debugging and user feedback.

The integration of visual feedback through viewport capture in BONCP introduces a novel approach to AI-BIM interaction. This capability enables AI assistants to understand spatial context and geometric relationships that pure textual descriptions cannot convey. The implementation includes intelligent image compression and optimization algorithms that balance visual quality with transmission efficiency.

The comprehensive resource system implemented in BIMCP demonstrates advanced data abstraction techniques that transform complex BIM models into queryable, AI-accessible information sources. The hierarchical URI scheme provides intuitive navigation while maintaining extensibility for future enhancements.

6.5 FINAL REMARKS

This research establishes a new paradigm for human-computer interaction in the construction industry, demonstrating that natural language interfaces can successfully bridge the gap between human design intent and complex technical software systems. The successful implementation of BIMCP and Kolektor-BONCP proves that Model Context Protocol provides a viable foundation for industry-wide adoption of AI-assisted design and construction workflows.

The fundamental shift from command-based interfaces to conversational interaction represents more than a technological advancement—it makes sophisticated building design and analysis capabilities accessible to more people. By eliminating the need for specialized programming knowledge, these

systems enable broader participation in digital construction workflows, potentially transforming how the industry approaches skill development and knowledge transfer.

The implications extend beyond individual productivity improvements to systemic industry transformation. As natural language interfaces become standard, the traditional barriers between technical and non-technical stakeholders diminish, enabling more integrated project teams and improved communication across disciplines.

The open-source nature of the MCP protocol and the research implementations provides a foundation for continued innovation and community-driven development. Rather than proprietary solutions that lock organizations into specific vendors, this approach enables competitive development while maintaining interoperability standards.

The research methodology successfully combines theoretical protocol development with practical implementation and real-world validation. The deployment at Kolektor Koling provides concrete evidence that these systems can deliver measurable business value while maintaining reliability and usability standards required for production environments.

The convergence of advanced AI capabilities, improved computational infrastructure, and industry recognition of digitalization needs creates an opportune moment for widespread adoption of AI-BIM integration. The construction industry's increasing comfort with digital tools positions these innovations for rapid acceptance and deployment across diverse organizational contexts.

The research establishes both technical foundations and practical pathways for continued development in AI-BIM integration. The challenges identified provide clear direction for future research efforts, while the successes demonstrated validate the fundamental approach. As AI capabilities continue advancing and construction industry digital adoption accelerates, the frameworks and methodologies presented in this research will support increasingly sophisticated applications.

The vision of natural, conversational interaction with building design systems is not only achievable but practical and valuable in real-world applications. The successful integration of Large Language Models with Building Information Modeling environments through the Model Context Protocol represents a significant step toward more intuitive, accessible, and powerful tools for creating and managing the built environment. As the construction industry continues its digital transformation, these AI-assisted workflows will play a central role in shaping how buildings are conceived, designed, constructed, and operated.

7 REFERENCES

[1] G. Lee, S. Jang, and S. Hyun, "A Generalized LLM-Augmented BIM Framework: Application to a Speech-to-BIM system," Sep. 26, 2024, *arXiv*: arXiv:2409.18345. doi: 10.48550/arXiv.2409.18345.

[2] C. Du, S. Esser, S. Nousias, and A. Borrmann, "Text2BIM: Generating Building Models Using a Large Language Model-based Multi-Agent Framework," Aug. 15, 2024, *arXiv*: arXiv:2408.08054. doi: 10.48550/arXiv.2408.08054.

[3] S. Wu, Q. Shen, Y. Deng, and J. Cheng, "Natural-language-based intelligent retrieval engine for BIM object database," *Comput. Ind.*, vol. 108, pp. 73–88, Jun. 2019, doi: 10.1016/j.compind.2019.02.016.

[4] M. Yin, L. Tang, C. Webster, S. Xu, X. Li, and H. Ying, "An ontology-aided, natural language-based approach for multi-constraint BIM model querying," Mar. 27, 2023, *arXiv*: arXiv:2303.15116. doi: 10.48550/arXiv.2303.15116.

[5] S. Jang, G. Lee, J. Oh, J. Lee, and B. Koo, "Automated detailing of exterior walls using NADIA: Natural-language-based architectural detailing through interaction with AI," *Adv. Eng. Inform.*, vol. 61, p. 102532, Aug. 2024, doi: 10.1016/j.aei.2024.102532.

[6] Y. Wei, X. Li, and F. Petzold, "Text-to-structure interpretation of user requests in BIM interaction," *Autom. Constr.*, vol. 174, p. 106119, Jun. 2025, doi: 10.1016/j.autcon.2025.106119.

[7] S. Shin, C. Lee, and R. R. A. Issa, "Framework for Automatic Speech Recognition-Based Building Information Retrieval from BIM Software," pp. 992–1000, Nov. 2020, doi: 10.1061/9780784482865.105.

[8] "Framework for Automatic Speech Recognition-Based Building Information Retrieval from BIM Software | Proceedings | Vol , No." Accessed: Jul. 11, 2025. [Online]. Available: https://ascelibrary.org/doi/10.1061/9780784482865.105

[9] M. Locatelli, E. Seghezzi, L. Pellegrini, L. C. Tagliabue, and G. M. Di Giuda, "Exploring Natural Language Processing in Construction and Integration with Building Information Modeling: A Scientometric Analysis," *Buildings*, vol. 11, no. 12, Art. no. 12, Dec. 2021, doi: 10.3390/buildings11120583.

[10] A. Shamshiri, K. R. Ryu, and J. Y. Park, “Text mining and natural language processing in construction,” *Autom. Constr.*, vol. 158, p. 105200, Feb. 2024, doi: 10.1016/j.autcon.2023.105200.

[11] K. Kim, M. Ivashchenko, P. Ghimire, and P.-C. Huang, “Context-Aware and Adaptive Task Planning for Autonomous Construction Robots Through Llm-Robot Communication,” May 14, 2024, *Social Science Research Network, Rochester, NY*: 4827728. doi: 10.2139/ssrn.4827728.

[12] D. P. Ghosh, *Agentic Ecosystem in Engineering Design: A Framework for Interoperable Legacy Tools and Emergent Collaboration via MCP/A2A Protocols*. 2025. doi: 10.13140/RG.2.2.27720.64008.