POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Automated Code Generation for IoT Edge Computing Solutions using AI

Advisors
Prof. Guido Albertengo

Candidate
Salvatore Cabras

July 2025

Abstract

The rapid evolution of Internet of Things (IoT) technologies has increased the demand for efficient and scalable solutions capable of handling data processing at the edge. Edge computing is the key to reducing latency, using less bandwidth, and enabling real-time decision making. While edge computing offers many advantages, the development and maintenance of software for IoT edge devices, such as those based on the ESP32 microcontroller and the ESP-IDF framework, remains a complex and time-consuming task. This thesis explores the use of AI-driven techniques for the automated generation, modification, and validation of embedded code for IoT edge computing environments.

The principal contribution of this work is the design and implementation of a multi-agent chat system using Microsoft's AutoGen-AgentChat framework. This system allows collaborative interactions between different AI agents, where each agent is responsible for a specific assignment in the software development pipeline, such as requirements interpretation, project organization, code generation, testing, and fixing code. The agents operate in a coordinated chat session, simulating a task-oriented dialogue that iteratively refines and validates the solution according by user definitions.

In the initial phase of code generation from scratch, the system employs a Retrieval-Augmented Generation (RAG) approach. RAG is a hybrid method that combines information retrieval with generative AI to ground the output in relevant documentation. An agent formulates queries based on the user's task description and retrieves useful content from a custom-built documentation database, including code snippets, which then serve as contextual input for code generation.

Three main workflows were implemented: (i) the generation of an ESP-IDF project from scratch based on a user-defined IoT task, (ii) the modification of a previously AI-generated project to reflect updated requirements, and (iii) the adaptation of an existing, manually developed ESP-IDF project.

For each workflow, there is an effective build step, where the system iteratively finds and fixes compilation or configuration errors iteratively until a successful build is achieved. This guarantees that the generated or modified code is not only syntactically correct, but also feasible and deployable on the target hardware.

The distinction between the second and the third workflow, edit for generated project and edit for existing project, is driven by an inherent architectural limitation: projects, for example those present in companies, are typically large and complex, making it unfeasible to provide their full context in a single chat message due to token constraints. In contrast, AI-generated projects follow a predictable structure that allows for direct and efficient interaction. To overcome this limitation in existing codebases, a separate workflow was introduced, based on search-driven heuristics. This approach analyzes function headers and file organization to identify the most relevant source files for a given modification, enabling targeted and scalable updates without overloading the conversational system.

While the first two workflows yielded consistently successful builds and coherent be-

havior, the third presented greater challenges due to the size and variability of input data. These were effectively mitigated by the use of heuristic-driven file selection, which enabled the system to handle large projects within the limits of the current AI model capabilities.

The system was implemented in Python and tested using ESP-IDF version 4.4.8 for the ESP32 platform. Through experimentation, the multi-agent, RAG-enhanced approach showed clear potential for streamlining embedded software development at the edge. Each project given by the agent chats was tested on an externally programmed basic ESP32 board.

In considering future work, some interesting enhancements include exploring dynamic embeddings for improved file selection for bigger projects, integrating agents specialized in advanced testing, supporting the use of custom third-party libraries, and coming up with cleverer strategies for identifying relevant portions of code to modify. These enhancements can lead to improve scalability, reliability, and usability in real-world IoT development workflows. It should also be interesting to see how it would work with the same structure with the same agents for another language with different purposes, loading data related to that language in order to use the RAG system.

This study aims to contribute to the growing field of AI-assisted software engineering, offering a flexible and extensible framework to automate embedded project generation and modification in resource-constrained environments. It also hopes to promote the application of AI agent-based systems to practical software engineering processes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, the Internet of Things (IoT) has become increasingly widespread, especially in environmental monitoring, smart homes, and many other applications. It has also found great importance in industry, where it is mainly used for real-time monitoring of production lines, predictive maintenance, and optimization of operating costs. The minimization of hardware size and the increase in low-power computational power have allowed some computational capabilities to be moved directly onto the device, giving rise to the Edge Computing paradigm.

Edge Computing is a fundamental piece of distributed systems design. Unlike centralized models that require continuous data sending to the cloud for processing, edge processing enables faster local decisions, reduced latency, improved privacy, and more efficient use of bandwidth, as illustrated in Figure 1.1 [5]. However, this decentralization leads to new technical challenges, mostly in the software field.

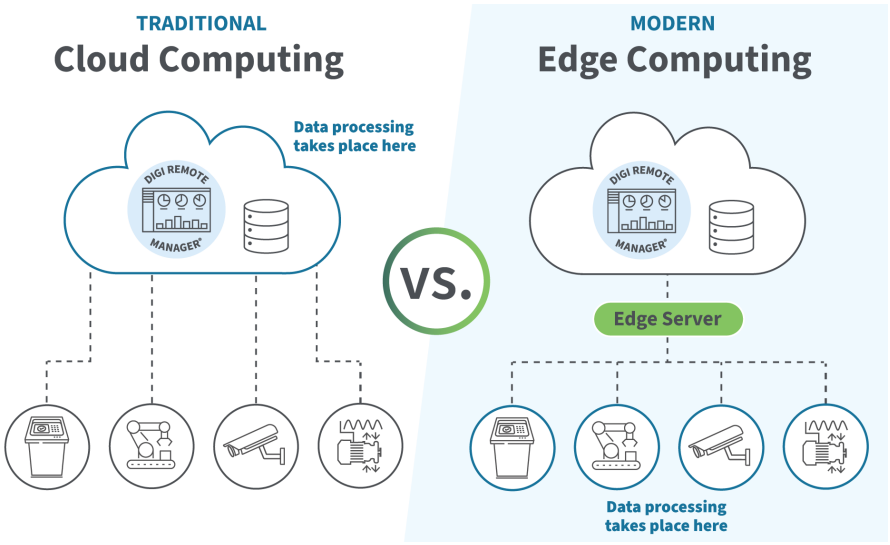

Figure 1.1: Comparison between Edge Computing and Cloud Computing. Source: [5]

Developing software for edge devices such as microcontrollers requires a deep knowledge of both the hardware and its associated development framework. In particular, ESP32 is one of the most widely used IoT (Internet of Things) applications. Its widespread use is due

to its optimal balance between cost, power, and connectivity. Nevertheless, working with tools such as the ESP-IDF (Espressif IoT Development Framework) can be challenging and complex for developers with little embedded systems experience, even though it is a powerful framework.

Concurrently, we are witnessing remarkable progress in Generative Artificial Intelligence (GenAI) technologies. Tools including GPT-4, GitHub Copilot, and Codex are revolutionizing the way we write code, document software, and design systems. Still, the application of these tools in embedded contexts, where resources are limited and architectures are constrained, can be better explored.

The ability to fully automate the generation of projects for IoT devices holds revolutionary promise: speeding up development, reducing human errors, and supporting intelligent software maintenance and evolution. But code generation alone cannot be enough, so we need a system that can also be able to verify, debug, and adapt code based on real-world constraints such as hardware compatibility, pin configuration, memory management, and other aspects typical of embedded software.

Within this context, the present thesis explores the implementation of a bot chat that leverages multi-agent AI system based on AutoGen-AgentChat, designed to autonomously generate, modify, and test software projects targeting the ESP32 platform. The adoption of a multi-agent conversational strategy, where each agent assumes a distinct role within the software development lifecycle (ranging from project manager to code fixer), constitutes a step toward a more collaborative and modular approach to automation in software engineering.

The system also integrates a component based on Retrieval-Augmented Generation (RAG), which allows to combine the capabilities of generative Artificial Intelligence with the precision of technical documentation retrieved in real time, reducing the risk of factual errors produced by the model.

Based on these considerations, the present work aims to investigate and develop a solution that combines AI capabilities with embedded system design. The next section (Section 1.2) outlines the specific objectives pursued during the project.

1.2 Objectives

The main objective of this thesis is to design and implement a system that facilitates the development of embedded software on IoT platforms, using artificial intelligence to support and automate the development process.

This general goal has led to the achievement of several intermediate goals, each of which represents a key step in the building of a complete, functional, and extensible AI-assisted development project.

First, a necessary point was to become familiar with the ESP32 microcontroller and its official development environment, the Espressif IoT Development Framework (ESP-IDF). This included learning how to configure, compile, and deploy embedded applications and understanding the standard structure and constraints of firmware development for resource-constrained devices.

A second step of the work was to investigate how generative AI models could be guided to autonomously produce valid and functional ESP-IDF projects. This required studying prompt engineering techniques, understanding the limitations of large-language models (LLM), and analyzing how to ground their output in relevant technical documentation.

To improve reliability and domain adherence, a data retrieval system was integrated that allows agents to consult a database of ESP-IDF documentation and code snippets,

aiming to generate grounded responses and reduce inconsistencies in code generation.

Since a planning choice was to use a multi-agent chat, the central goal focused on designing multi-agent architectures where each agent could specialize in a specific stage of the software development pipeline, such as retrieve documentation, organize project, code generation, validation, and others, within a structured, goal-oriented chat environment. So, three different multi-agent chats were implemented; we will go in detail in Chapter 3.

Especially for large existing projects, token limits posed a major challenge. A dedicated goal was to design strategies (such as source code filtering and function header analysis) that allow for context-aware modifications without exceeding the input capacity of the model.

The expected result is an extensible and practical project that accelerates the software development cycle in the field of embedded IoT, opening new possibilities in the field of AI-assisted software engineering.

Chapter 2

Background

2.1 Internet of Things and Edge Computing

2.1.1 Overview of the Internet of Things

In last years, the Internet of Things (IoT) has become increasingly important in the fields of information technology, engineering, and industrial automation. The term IoT refers to a network of interconnected devices that can collect, transmit, and process data through network connections. There are various types of devices that can take advantage of IoT technologies, for example, environmental sensors, smart home appliances, complex industrial systems, security cameras, wind turbines, and many others. They contribute to the creation of a distributed ecosystem capable of interacting with the real world in real time.

A typical architecture of an IoT system is shown in Figure 2.1, where multiple devices are connected through local networks to edge gateways, which in turn communicate with cloud infrastructures. This layered design enables flexibility in data processing and decision making.

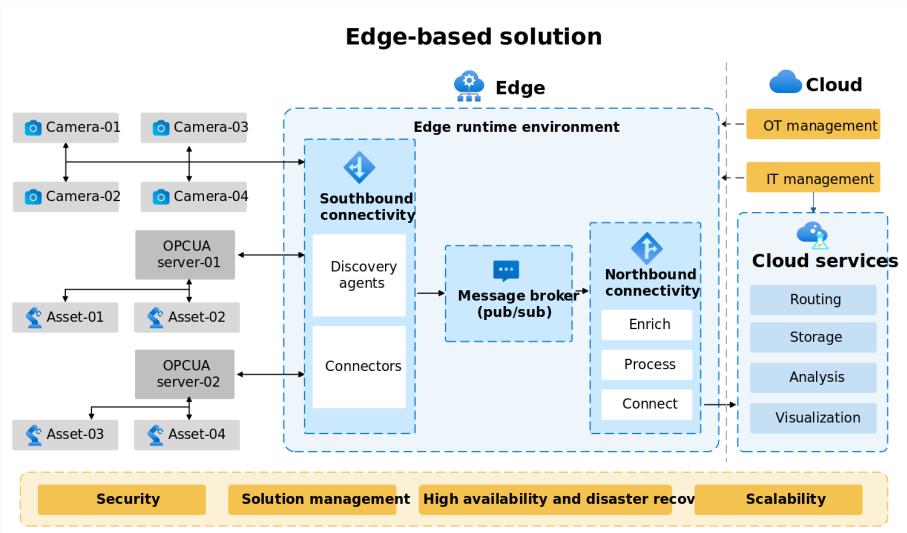

Figure 2.1: Typical architecture of an IoT system with edge computing. Source: [19]

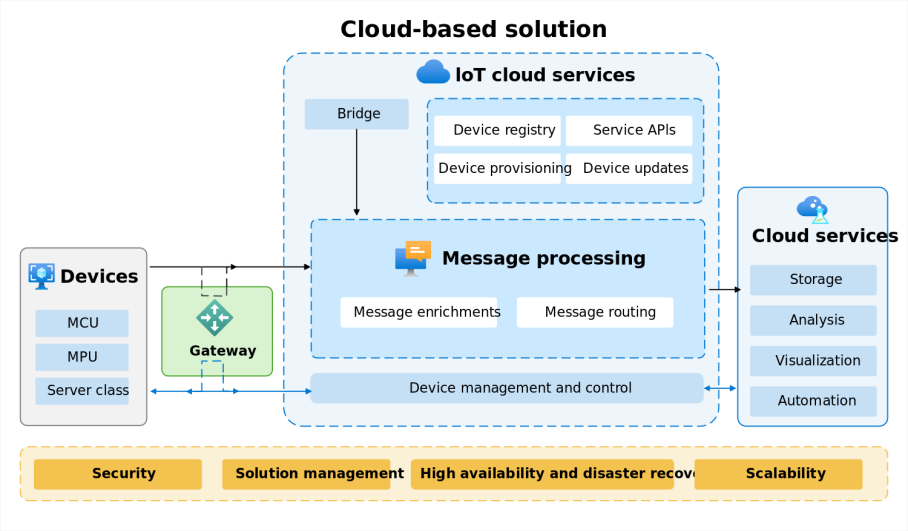

Figure 2.2: Typical architecture of an IoT system with cloud computing. Source: [19]

Otherwise, as we can see in Figure 2.2, a cloud-centric architecture places most of the computational burden on centralized cloud services, relying on the constant transmission of raw data from edge devices. This can lead to increased latency, bandwidth usage, and potential privacy concerns.

According to the *State of IoT Summer 2024* reported by IoT Analytics, the number of connected IoT devices reached 16.6 billion by the end of 2023, marking a 15% increase compared to 2022. This amount is expected to grow by an additional 13% to 18.8 billion devices by the end of 2024 [13]. Despite macroeconomic challenges and geopolitical tensions, the growth trajectory remains strong, driven by increasing adoption by enterprises. This trend highlights the growing need for efficient and scalable solutions at both the hardware and software level to support this ongoing digital transformation.

The progressive miniaturization of hardware and the improvement in energy efficiency have made possible the widespread diffusion of embedded devices equipped with network connectivity. This has expanded the applicability of IoT in sectors like precision agriculture, home automation, smart cities, healthcare, and industry 4.0.

As mentioned above, the traditional model, based on centralized processing in the cloud, has several limitations, particularly in terms of latency, bandwidth consumption, and privacy management. To address all these issues, the **Edge Computing** paradigm has emerged, which proposes an inversion of the processing flow: data are processed as close as possible to its origin, i.e. directly on board the device or in proximity to it (for example on local gateways).

2.1.2 Features and advantages of Edge Computing

The Edge paradigm introduces a series of benefits that make it particularly suitable for IoT scenarios:

- **Latency reduction**: The processing of local data removes the need to send information to remote servers for computation, enabling real-time responses that are essential for critical applications, such as incident detection and robotics.
- **Less bandwidth consumption**: By limiting the amount of data transmitted to the cloud, network traffic will be significantly reduced, which in turn improves the

scalability of the system and overall performance.

- **Privacy e security**: Keeping all sensitive data locally helps prevent transmission over external channels, reducing the risk of potential data breaches and unauthorized access.

- **Resilience**: In spite of a network disconnection, the devices are able to continue to operate independently at a local level, maintaining the same functionalities without interruption.

It is worth to notice that the edge paradigm does not exclude the cloud, but rather make use of it as a synergistic extension of it. In hybrid architectures, often referred to as *cloud-edge continuum*, edge devices pre-process data by filtering relevant information or aggregating it before transmitting it to the cloud for more complex analysis or long-term storage. This balanced approach enables the optimization of the overall system by carefully managing and improving key factors such as operational costs, computational performance, and reliability. By smartly distributing workloads between edge and cloud components, it ensures more efficient resource utilization, reduced latency, and increased system robustness, which are very critical for modern, data-intensive applications.

2.1.3 Types of edge nodes

In the Edge context, it is possible to distinguish different levels of devices based on their computational capabilities, energy constraints, and functional roles within the entire system architecture. This classification is needed to understand how processing tasks are distributed across the network and to design efficient and scalable solutions that suit the specific characteristics of each device level.

- **Endpoint devices (or leaf nodes)**: These include sensors, actuators, and low-power microcontrollers, for example, ESP32, STM32, or nRF52. They are typically positioned closest to the data source and are responsible for data acquisition and initial signal processing. Due to their constrained hardware resources, they require highly optimized software and minimal power consumption.

- **Edge gateways**: These are more powerful embedded systems (e.g., Raspberry Pi, NVIDIA Jetson Nano) that serve as intermediaries between endpoint devices and higher-level computing resources. They aggregate data from multiple endpoints, perform local pre-processing or filtering, and may manage communication with remote servers. Their enhanced computational capabilities enable more complex tasks to be executed closer to the data source, reducing latency and network load.

- **Edge cloud or cloudlets**: These are small-scale localized server instances located near the end devices, often within the same local area network. Designed to handle computationally intensive tasks with low latency, they offer a compromise between centralized cloud computing and fully distributed edge architectures.

This thesis focuses mainly on the first category of devices, endpoint nodes, where severe hardware limitations introduce significant challenges to automating the software development process.

2.1.4 Software development challenges in edge environments

Software development for devices in edge environments requires a radically different approach compared to traditional desktop or server programming. Strict constraints demand careful attention to every detail, from power consumption and direct memory management to interacting with hardware peripherals through registers or drivers. All of these factors make the development environment closer to electronic engineering than to general-purpose software engineering.

On one hand, edge computing offers new opportunity, on the other hand, introduce new complexity, in particular in the context of software develpment. Edge devices, such as microcotrollers, are subject to strict constraints in terms of:

- **Limited hardware resources** CPU, RAM, memory space;
- **Manual management of configuration and build systems**, often requiring custom toolchains, linker scripts, and platform-specific settings;
- **Lack of dynamic runtime environments**, requiring low-level programming in languages like C or C++;
- **Complexity in managing hardware peripherals** and low-level communication.

These aspects imply the need for specialized skills that often represent a barrier to entry for developers without experience in embedded systems. For this reason, there has been growing interest in recent years in automated and AI-assisted solutions aimed at simplifying the development cycle in such environments.

2.1.5 Microcontrollers and development frameworks

A significant example of an edge device platform is the ESP32, Figure 2.3 represents the board block diagram, a microcontroller designed by Espressif Systems, which features Wi-Fi and Bluetooth connectivity and is particularly popular in the IoT field for its excellent quality and efficiency, versatility, and support of a large open source community. The ESP32 is based on a Tensilica Xtensa LX6 or LX7 dual core architecture, with clock frequencies up to 240 MHz, support for advanced peripherals (SPI, I2C, ADC, DAC, PWM), and integrated flash memory.

The ESP-IDF (Espressif IoT Development Framework) is the official framework for firmware development on the ESP32, an SDK that provides a complete environment for writing, compiling and deploying C / C++ code [6]. ESP-IDF also includes some libraries useful for networking, peripheral management, cryptography, over-the-air updates (OTA), and support for FreeRTOS as a real-time operating system. It is designed to offer flexibility and performance, but at the expense of greater complexity than other more accessible environments such as Arduino IDE or PlatformIO.

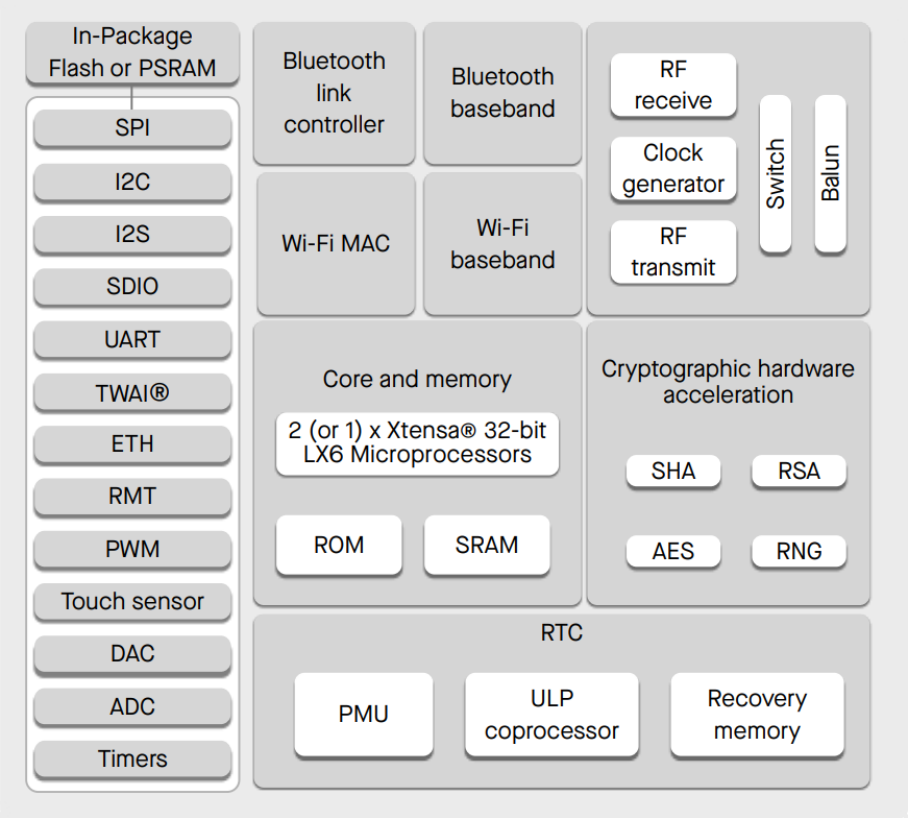

Figure 2.3: ESP32 functional block diagram. Source: [7]

Despite its potential, ESP-IDF does require a certain learning curve, as it involves familiarity with tools such as `CMake`, `idf.py`, and the use of specific embedded toolchains for the target processor. Installing the toolchain, configuring project files, managing dependencies, and understanding the organization of software components are all nontrivial, especially for beginner developers.

Within this scenario, the possibility of exploiting generative artificial intelligence to support and automate repetitive or complex tasks in embedded software development emerges. The adoption of advanced linguistic models can simplify the creation of boilerplate code, the initial configuration of the project, and the integration of complex components, such as TCP/IP stacks, MQTT protocols, or sensor management.

In light of these complexities, AI-based approaches are increasingly being explored to support developers in tasks such as code configuration, generation, and validation. The idea of building systems capable of interacting in natural language, understanding requirements, and generating correct code that complies with framework specifications becomes particularly attractive in such a technical and fragmented context. This thesis contributes to this emerging field by proposing a multi-agent architecture that aims to streamline and automate the development process on the ESP32 platform.

2.2 AI-assistance in Software Development

Generative artificial intelligence (GenAI) has significantly reshaped the way software is designed, written, and maintained. Thanks to advances in Large Language Models (LLMs) such as *GPT-4*, *Codex*, *Claude*, and many others, it is now possible to develop code more

quickly, accurately, and with greater assistance, reducing a lot of time and costs associated with traditional software development.

2.2.1 The evolution of development tools

The models mentioned before do more than just complete code fragments, they can understand the semantic context of an entire code base, answer technical questions, generate complete functions, explain complex algorithms, and even write technical documentation. Basically, they act as real *virtual assistants* within the software development cycle, helping to reduce both the time and cost traditionally required for building and maintaining software systems.

2.2.2 LLM-based tools for programming

One of the most well-known examples is **GitHub Copilot**, it's also one of the most used, developed by *GitHub* in collaboration with *OpenAI*, which uses models based on *Codex* to suggest code in real time that is consistent with what the developer is writing. Direct integration with the most diffused IDEs (Integrated Development Environment), like *Visual Studio code*, provides a smooth and interactive experience.

GitHub studies indicate that using *Copilot* can lead to an increase in productivity of up to 55% in repetitive tasks or when writing boilerplate code [21]. More recent academic studies [20] have also confirmed similar results, showing an average productivity improvement of 50% on using *Github Copilot*, results reported in Table 2.1.

Table 2.1: Comparison of Development Efficiency with and without AI Copilot [20]

Metric	**Without Copilot**	**With Copilot**
Task Completion Speed (relative)	100%	155.8%
Code Quality Improvement	0%	70%
Debugging Time Reduction	0%	70%
Developer Productivity (relative)	100%	155.8%
Innovation Potential (scale 1-10)	7	9
Accessibility for Novice Developers (scale 1-10)	5	8

Another comparative analysis, reported in the same study [20], evaluates the efficiency gap between traditional development environments and AI-powered IDEs, as shown in Table 2.2.

Table 2.2: Performance Comparison: Traditional IDEs vs. AI-Enhanced IDEs [20]

Feature	**Traditional IDE**	**AI-Enhanced IDE**
Bug Detection	100%	160%
Coding Speed	100%	130%
Documentation Coverage	70%	95%
Code Refactoring Efficiency	80%	110%
Version Control Insights	85%	115%
CI/CD Pipeline Issue Detection	90%	120%
Code Review Efficiency	75%	105%
Overall Developer Productivity	100%	135%

Other commonly used tools are:

- **Amazon CodeWhisperer**, offered by Amazon Web Services (AWS), which provides optimized contextual suggestions for cloud services and languages like *Python*, *Java*, and *JavaScript*.
- **Tabnine**, uses lighter, language-specific models with a focus on data privacy and on-premise deployment, making it suitable for corporate environments with strict policies.
- **Cursor**, a conversational IDE built around *GPT-4*, which allows users to ask natural language questions about their code, receive contextual modifications, or perform interactive refactoring.
- **Codeium**, combines AI-based code completion and semantic search tools within code repositories.

These tools not only accelerate code writing, but also support debugging, testing, optimization, and refactoring, redefining the development cycle as a continuous collaboration between developer and AI.

2.2.3 Current limitations in the embedded context

Despite the promising results obtained in the development of general-purpose software, the use of generative AI in the *embedded programming* domain and in particular in the *IoT* and *Edge Computing* context is still relatively limited. Embedded environments pose particular constraints:

- Heterogeneous and often poorly documented hardware architectures.
- Serious restrictions on memory, power consumption, and response time.
- Specific toolchains and non-standardized workflows.

- Need to integrate code with drivers and low-level components.

All of these factors make it difficult to use generalist LLMs directly. To be truly effective, AI tools must adapt to the specific domain, understand the technical context, and interface with low-level documentation.

2.2.4 Towards Multi-Agent Conversational Systems

To deal with these challenges, more advanced solutions emerge, which exploit the use of **conversational multi-agent** architectures. In these systems, as in a developer group, each agent can be specialized in a phase of the development process, a simple example can be:

- **Agent-1** Interpreting requirements in natural language.
- **Agent-2** Retrieval of relevant technical documentation (with RAG techniques).
- **Agent-3** Code generation compliant with the specific framework
- **Agent-4** Automatic validation and iterative code correction.

Different from simple completion tools, these architectures promote structured interaction, where AI can reason more purposefully, collaborate on complex tasks, and dynamically adapt to project needs.

In line with this evolution, the next section will explore an even more advanced paradigm, that is the use of LLM-based multi-agent systems, where multiple specialized agents cooperate within a conversational framework to address the different phases of software development in a coordinated way. These approaches represent a step beyond passive assistance, introducing more structured and scalable forms of AI-driven collaboration, particularly promising in the embedded and IoT context.

2.3 AutoGen and Multi-Agent Systems

2.3.1 Introduction to multi-agent system

The **multi-agent systems** (MAS) represent a branch of distributed artificial intelligence where there are more than autonomous entities, called *agents*. They cooperate and interact with each other to achieve a common goal. In the context of LLMs, this paradigm has evolved into systems in which multiple agents, each with different roles and skills, collaborate through a conversational interface to solve complex tasks, highly similar to a chat.

These systems transform the interaction with AI from a single user-model exchange to a **multi-turn collaborative dynamic**, in which agents give information, make decisions and coordinate actions. This approach has proven to be particularly effective for multi-step tasks, such as writing code, generating software projects, running data pipelines, complex document processing, summarize text, and many other tasks.

2.3.2 Conversational Architectures Based on LLMs

In a multi-agent conversational system, each agent can be equipped with:

- a **profile** (e.g., programmer, analyst, tester, information retrieval,...),

- a **behavior prompt** it says what it should do and what its objective is,
- a **set of functions** functions that can be called by an agent that can be useful to complete some task, like searching information online.

The dialogue between agents is orchestrated by a **controller**, *manager* or *planner*, which establishes turn-taking rules, defines exit conditions and supervises the consistency of the interaction, usually it calls an agent saying what it should do. This architecture allows each agent to:

- Access external tools, for example, call APIs and script execution.
- Write and execute code.
- Query resources via Retrieval-Augmented Generation (RAG).
- Analyze, validate, and improve the work of other agents.

2.3.3 Main frameworks e libraries

In recent months, different open-source libraries have emerged that make the development of LLM-based multi-agent systems accessible. The most relevant are the following.

AutoGen

AutoGen [16] is a library developed by Microsoft Research that allows to build a multi-agent chat based on LLM. Its central component is the high-level AgentChat API, which provides an intuitive framework to build complex interactions between agents. While, AgentChat is the recommended starting point for novice users, advanced users can leverage the event-driven programming model of `autogen-core` for greater flexibility and control.

The core of the AgentChat interaction is the AgentChat module, which allows you to define agents with specific roles, tools, memory capacity, and well-defined goals. These agents interact by communicating through a shared chat context.

Through a simple but flexible setup, you can orchestrate conversations between different entities:

- **LLM Agents**: based on large language models (LLMs), including those from OpenAI, Azure OpenAI, Azure AI Foundry, Anthropic), Ollama, and accross the Semantic Kernel adapter
- **Human Users** mainly represented by the *UserProxyAgent*, this is a built-in agent that acts as a proxy for the user, allowing them to provide feedback to the team during a run or to initiate new interactions.
- **Wrappers for external tools**: agents can use functions or API calls as tools. This allows them to perform specific actions such as web search, data analysis, or interaction with files and systems.

Each agent in AgentChat is built individually and requires the declaration of key attributes:

- **Name**: the name of the agent, it is advisable to choose meaningful names (in instance, *PlanningAgent, CodeGeneratorAgent, FileOrganizerAgent, ecc..*) so that other agents and the model understand what its purpose is.

- **Description**: a very brief description that says which agent is specialized. This description is crucial, especially in setups like *SelectorGroupChat*, where the model uses it to determine the most appropriate agent for the next round.
- **Model Client**: it defines the configuration of the model, it has other fields like *model, api_version, azure_endpoint, api_key and azure_deployment* which define the model (gpt-4o), the type of AI, the version and the endpoint with a valid key. AutoGen supports various clients for popular providers, and you can also extend them or create custom ones.
- **Tools**: these are functions that can be called by the agent to perform specific actions. The *AssistantAgent* is preconfigured to use tools and automatically converts Python functions to *FunctionTool* by generating a schema from their arguments and docstrings. When an agent runs a tool, its output is returned as a string in a *ToolCallSummaryMessage*. If the output is not well-formed, you can set `reflect_on_tool_use=True` in the agent to allow the model to summarize the tool's output. Models can also support parallel tool calls.
- **System Message**: this is the initial prompt that will make the agent understand who he is and what his purpose is within the chat. So, it is a key part, here you must clearly specify everything he has to do, because it creates a bias to the agent.

To create a chat, you need to assemble a multi-agent team that, as said before, is a group of agents that work together to achieve a common goal. In a team, the agents you want to interact with have to be called. The AgentChat system supports different team configurations, see Figure 2.4(*bottom-right*):

- **RoundRobinGroupChat**: a group chat setup in which agents take turns speaking in a round-robin sequence. Each agent broadcasts their response to all the others, maintaining a consistent context.
- **SelectorGroupChat**: a configuration where the next agent to speak is dynamically selected after each message from a *ChatCompletion* model. The selection is based on the context of the conversation and the agent descriptions. This allows for dynamic and context-aware collaboration.
- **MagenticOneGroupChat**: a general-purpose multi-agent system designed to handle open-ended tasks involving web and file-based resources across multiple domains. Magentic-One is implemented as an AgentChat team and leverages a leading Orchestrator agent for high-level scheduling, task delegation, and progress tracking. It includes specialized agents such as MultimodalWebSurfer (for web browsing), FileSurfer (for reading local files), Coder (for writing code), and ComputerTerminal (for executing code).
- **Swarm**: a team setup in which agents delegate tasks to other agents based on their capabilities, using special HandoffMessage messages. All agents share the same message context, allowing local decisions about task scheduling, rather than relying on a central orchestrator. An AssistantAgent can specify who it can pass control to through the handoffs argument.

In Figure 2.4 are shown different implementations using multi-agent conversations. AutoGen agents, as said above, are customizable and can be based on LLMs, tools, humans, and even a combination of them (*left*).

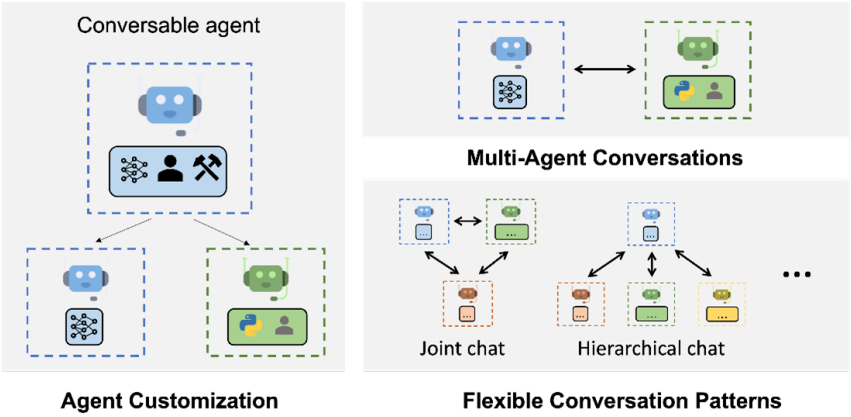

Figure 2.4: Example of different workflows. Source: [17]

Additional features and key concepts

- **Type of Messages**: AgentChat handles communication via messages, which are divided into two main categories. **Agent-Agent Messages**, used for communication between agents, such as TextMessage and MultiModalMessage and **Internal Events** that are internal messages to the agent, such as *ToolCallRequestEvent* and *ToolCallExecutionEvent*, these are often contained in the `inner_messages` attribute of a Response.

- **Structured Output**: AgentChat allows you to generate structured JSON output using Pydantic BaseModel classes, for models that support it. This is also useful for incorporating Chain-of-Thought reasoning into agent responses.

- **Streaming Messages**: it is possible to view messages as they are generated by agents using the `on_messages_stream()` or `run_stream()` method. Additionally, by setting `model_client_stream=True` in `model_client`, individual generated tokens can be streamed to the model as they arrive.

- **Memory**: for memory management, AutoGen offers a memory protocol to maintain a store of useful facts that can be smartly added to an agent's context prior to an action. This supports patterns such as RAG (Retrieval-Augmented Generation). The protocol includes methods such as `add` (add entries), `query` (retrieve relevant information), and `update_context` (change the internal context of the agent). Examples include *ListMemory* (simple list-based memory) and *ChromaDBVectorMemory* for integration with vector databases for more complex memories.

- **Managing State**: it is possible to save the state of agents and teams to disk (e.g. in JSON format) and reload them for persistent sessions. The `save_state()` and `load_state()` methods are provided for this purpose. This is especially useful in web applications with stateless endpoints.

- **Custom Agents**: they can be built by inheriting from the *BaseChatAgent* class and implementing the abstract methods `on_messages()` and `on_reset()`, it is worth when the default behaviors are not enough. You can also make custom agents declarative by inheriting from the Component class, allowing them to be serialized and deserialized via the `dump_component()` and `load_component()` methods.

- **Termination Conditions**: a crucial aspect in managing the duration of conversations. AgentChat provides several termination conditions that can be combined using logical operators `AND` and `OR`:
  - **MaxMessageTermination**: stops after a specified number of messages.
  - **TextMentionTermination**: stops when a specific text (like `TERMINATE`) is mentioned.
  - **HandoffTermination**: stops when a handoff is requested to a specific target.
  - **ExternalTermination**: allows external control of termination (e.g. from a UI).
  - **TextMessageTermination**: stops when a `TextMessage` is produced by an agent.
  - **FunctionCallTermination**: stops when a specific function call is executed.
- **Human-in-the-Loop**: in addition to interacting with *UserProxyAgent* during a blocking execution, you can provide feedback for the next execution by configuring `max_turns` or using termination conditions to pass control to the application/user.
- **Single-Agent Team**: useful for running an *AssistantAgent* in a loop until a termination condition is met, unlike its `run()` or `run_stream()` execution which stops after a single step.

In this work, AutoGen, version 0.4.9, has been used as a central framework to implement a collaborative chat between agents specialized in embedded software development. All teams are created using *SelectorGroupChat* and each agent is defined by the syntax of `agentchat` mentioned before, for tasks like:

- Parsing of technical documentation.
- C/C++ code generation for microcontrollers.
- Automatic testing and debugging.

In Chapter 3 the configurations used and the structure of the agents and teams created will be better explained.

CrewAI

CrewAI [4] is a Python framework designed to build collaborative teams of AI agents. It allows for orchestrating how multiple agents, each with their own specializations, work together to achieve a common goal. It is particularly suited to sequential tasks and orchestrating agent behavior in real-world software projects.

It works on the basis of a few key components:

- **Crew (Team)**: this is the high-level organization that manages the entire team of AI agents, overseeing workflows and ensuring collaboration to deliver results. It is the container that orchestrates the work.
- **Agents (AI Agents)**: similarly to AutoGen Agents, they are the specialized members of the team. Each agent is defined with a unique personality, a specific role, a clear goal, and a backstory that shapes its behavior. Agents are more effective when they are specialized, rather than generic. They can be equipped with custom or predefined tools to interact with external services and data sources, such as web search or data analysis.

- **Tasks**: these are the concrete work that agents are supposed to do. Each task has clear goals, detailed instructions, and expected outputs. It is important to note that task design is more critical to system success than agent definition itself. A well-designed task should have a single purpose and a well-defined output.
- **Process**: defines the collaboration model between agents and controls the assignment of tasks. Processes can be sequential, one agent finishes before the next one starts, or parallel, multiple agents work simultaneously.
- **Flows**: offer more granular control and structured, event-driven automations that can handle conditional logic and state transitions. Flows integrate seamlessly with Crews, allowing you to balance high autonomy with precise control over execution. They are used for deterministic processes or specific API integrations, while Crews are ideal for more open and collaborative problem-solving tasks.

In short, CrewAI enables to build systems where AI agents, with their specialized roles and capabilities, intelligently collaborate on complex tasks, following a defined process to achieve a common goal, just like a human team would.

MetaGPT

MetaGPT [22, 23] is an innovative meta-programming framework developed for multi-agent collaboration based on Large Language Models (LLM) [9]. It stands out for applying a direct analogy with the business organization, where each agent covers a classic and specialized role, like Product Manager, Architect, Project Manager, Engineer, QA Engineer. The goal is to introduce human practices into multi-agent frameworks.

Its functioning is based on the incorporation of Standard Operating Procedures (SOPs), which are encoded in sequences of prompts to define structured workflows and reduce errors, simulating a real assembly line. MetaGPT agents are equipped with human-like domain expertise and follow these SOPs to complete software development tasks consistently and autonomously.

Unlike other frameworks that may use unconstrained natural language communication, MetaGPT emphasizes structured communication interfaces, requiring agents to produce specific, formalized outputs, such as requirements documents, system diagrams, code. To facilitate information exchange and prevent overload, MetaGPT employs a shared message pool and a publish-subscribe mechanism. This allows agents to publish their own structured messages and transparently access only the information relevant to their role [9].

Another important feature is the *executable feedback* mechanism, a self-correcting feature implemented, for example, for the Engineer agent. After initial code generation, the agent runs unit tests and verifies runtime correctness, iteratively debugging the code until tests pass or a retry limit is reached. This approach significantly improves the quality of the generated code and reduces the cost of human review.

This framework has demonstrated state-of-the-art performance in software engineering benchmarks such as *HumanEval* and *MBPP*, and has shown a 100% task completion rate on its *SoftwareDev* dataset.

LangGraph

LangGraph [12] is a framework built on *LangChain* that allows you to define dynamic conversational graphs between agents. It is particularly useful for building modular pipelines and managing complex interactions between AI agents.

LangGraph has several main features:

- **Reliability and Controllability**: it enables guiding agent actions with moderation checks and human-in-the-loop approvals. LangGraph also maintains context for long-running workflows, ensuring agents stay on course.
- **Low-level and Extensible**: it allows for building custom agents with fully descriptive, low-level primitives, free from rigid abstractions that might limit customization. This facilitates the design of scalable multi-agent systems, where each agent serves a specific role tailored to its use case.
- **First-class Streaming Support**: it offers token-by-token streaming and streaming of intermediate steps, providing users with clear visibility into agent reasoning and actions as they unfold in real time.

LangGraph excels at enabling complex logical conditions, robust state management to maintain conversation context across calls, and flexible message routing, including the ability to route complex queries to a human for review. With LangGraph, it is possible to build support chatbots capable of answering common questions by searching the web, maintaining conversation state across calls, routing complex queries to a human for review, using custom state to control their behavior, and even rewinding to explore alternative conversation paths.

2.3.4 Agent Coordination and Cooperation Models

The orchestration of the AI agent [11] is the process of coordinating multiple specialized AI agents within a unified system to efficiently achieve shared objectives. When a single AI model or agent is insufficient for complex tasks, orchestration becomes essential to enable multiple agents to work together efficiently. Figure 2.5 points out a draft of the difference between the single-agent and multi-agent approach.

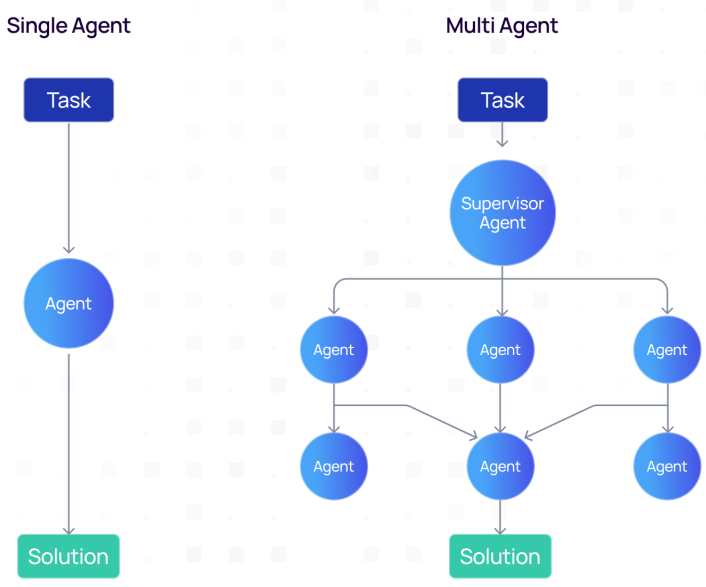

Figure 2.5: single vs multi agent. Source: [2]

The ways in which agents collaborate can be different, in Figure 2.6 diagrams are shown representing the types of orchestration.

- **Direct (or Decentralized) Orchestration**: in this model multi-agent systems operate through direct communication and collaboration among the agents themselves.
- **Centralized Orchestration**: a single orchestrator AI agent acts as the system's brain, directing all other agents, assigning tasks, and making final decisions, ensuring consistency and predictable workflows.
- **Hierarchical Orchestration** is a variant of centralized orchestration where AI agents are arranged in layers, with higher-level orchestrator agents overseeing and managing lower-level agents, balancing strategic control and task-specific execution. In MetaGPT, an Orchestrator agent can create a plan and delegate tasks to other agents, monitoring progress, and dynamically revising the plan.
- **Federated Orchestration**: this approach focuses on collaboration between independent AI agents or separate organizations. The distinguishing feature is that these agents work together without fully sharing data or giving up control over individual systems. This is especially useful in situations where privacy, security, or regulatory constraints prevent unrestricted data sharing. Industries such as healthcare, banking, or inter-company collaborations can greatly benefit from federated orchestration, as it allows them to combine the capabilities of different AI systems while maintaining sovereignty over their data and processes.

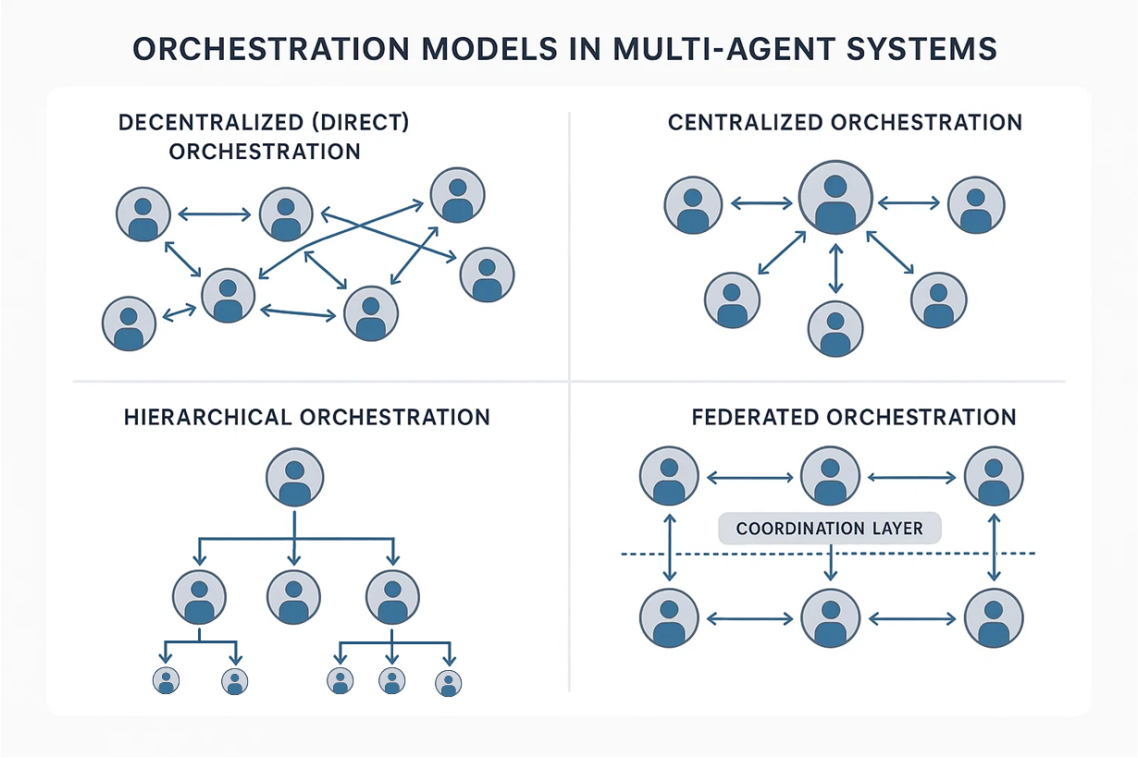

Figure 2.6: Different types of multi-agent systems orchestration

Some common patterns that a multi-agent model can have include:

- **Critic-Developer**: one agent writes code or generates a response, while another agent evaluates it and provides constructive feedback. In AutoGen [16], this pattern is implemented with *RoundRobinGroupChat*, where a `primary_agent` generates text and a `critic_agent` reviews it, requesting revisions until approval.

- **Planner-Executor**: one agent designs a strategy or breaks down a complex task, while other agents implement it. AutoGen's SelectorGroupChat uses a Planning Agent that breaks down complex tasks into subtasks and delegates them to specialized agents (Web Search Agent, Data Analyst Agent) [16].

- **Operating Procedures (SOPs)**: define a sequential workflow, a Product Manager analyzes requirements, an Architect creates the system design, a Project Manager distributes tasks, Engineers execute the code and QA Engineers test it[22].

- **Team Chat**: this pattern refers to conversations among multiple agents, where all share the same conversation context. AutoGen's AgentChat is specifically designed for building LLM-based multi-agent chats. In these teams, such as RoundRobinGroupChat or SelectorGroupChat, as mentioned in the previous section,agents interact by exchanging messages in a shared chat, maintaining a consistent context.

Regardless of the coordination model, modern frameworks such as AutoGen and CrewAI emphasize the creation of specialized agents with well-defined roles, goals, and backstories, the use of flexible tools to interact with external services, state management to maintain the context of the conversation, and streaming support for greater observability of agent actions. Additionally, the integration of a `human-in-the-loop` enables human intervention for feedback and approvals, improving the reliability and control of multi-agent systems.

2.3.5 Benefits of software development automation

The multi-agent approach to software development automation offers numerous benefits, enhancing the ability to tackle complex tasks more efficiently and reliably.

The multi-agent approach allows to:

- **Dividing complex tasks into independent subtasks**: frameworks facilitate breaking down an overall goal into smaller and more manageable steps.

- **Enabling self-correction and mutual review**: a significant strength is the ability of agent teams to iteratively improve their work. MetaGPT implements an executable feedback mechanism where the `Engineer` can run unit tests on generated code, receive feedback, and autonomously debug the code, reducing hallucinations and ensuring runtime correctness[23]. AutoGen [16], through teams like *RoundRobinGroupChat*, enables critic-developer patterns, where a critical agent provides constructive feedback and requests revisions until the result is approved. LangGraph [12] also offers the ability to steer agent actions with moderation controls and human-in-the-loop approvals, improving reliability.

- **Simulate realistic work processes**: these frameworks can replicate the organization of a human team.

- **Integrate heterogeneous capabilities (such as code, testing, search)**: agents are equipped with flexible tools that allow them to interact with external services and data sources. Large language models (LLMs) are typically limited to generating text or code. However, to tackle complex tasks, AI agent frameworks augment agents with the ability to use external tools that perform specific actions.

In the context of software development, this leads to:

- **Increased reliability of generated code**: for example, the adoption of SOPs, structured outputs, and executable feedback mechanisms leads to a significant increase in system robustness and a reduction in collaborative inefficiencies and logical inconsistencies.
- **Improved transparency in decision-making**: some frameworks make viewing available in real time the agent's *internal thought process*, like a chat, which conducts to the final answer, increasing the understanding of the agent's actions.
- **Scalability to multi-component projects**: AI agent orchestration is significant when a single AI model or agent is not sufficient to tackle complex tasks. It enables efficient collaboration of multiple agents to achieve a common goal, even in large-scale applications such as healthcare or finance.
- **Potential reduction in development time**: efficiency in agent orchestration can translate into reduced development time, and it also significantly reduces human review costs.

2.3.6 Multi-agent systems in embedded and IoT contexts

The use of multi-agent systems in the embedded and IoT context is still in an experimental stage, but promises interesting results, thanks to:

- **Ability to handle technical documentation** and datasheets autonomously: AI agent frameworks improve agents with the ability to use external tools that perform specific actions. This includes the ability to interact with APIs, data sources, and web search engines. Specific agents can navigate the web or browse local files to retrieve information. These agents are also capable of processing multimodal input, which is essential for understanding datasheets and complex documentation that often includes diagrams and tables.
- **C/C++ code generation for specific microcontrollers**: frameworks are designed for collaborative software engineering, enabling automatic code generation. Specialized agents are optimized to write code and analyze information. The process includes an executable feedback mechanism that allows agents to write and run unit tests, receive results, and iteratively debug the code. This significantly improves the reliability and accuracy of the generated code at runtime.
- **Virtual Team Simulation**: multi-agent systems employ a network of AI agents, each designed for specific tasks, that work together to automate complex workflows and processes, such as Firmware Engineer, Validator, and Tester. Frameworks allow you to create specialized agents with distinct roles, goals, and backstory. Agents interact by exchanging messages in a shared chat or through a global message pool, maintaining common context, and tracking progress. They can behave like a team that wants to develop a project.

- **Iterative support in configuring build systems and toolchains**: agents can be equipped with flexible tools that allow them to interact with external services and execute code. This is crucial for configuring and validating embedded development environments. The ability to self-correct via executable feedback allows agents to iterate on configurations and code, improving the quality and effectiveness of the final output.

Building on these principles, the thesis introduces an AutoGen-based system that coordinates AI agents in a collaborative process to develop embedded applications tailored for edge computing environments.

2.4 Retrieval-Augmented Generation (RAG)

2.4.1 Introduction to RAG

Large language models (LLMs) show remarkable generative capabilities, but are limited by the fact that their knowledge is static and embedded in the model weights, updated only during the training phase. To go over these limitations, especially in dynamic or specific contexts such as the embedded environment, the Retrieval-Augmented Generation (RAG) paradigm has assumed an important role. It was introduced as a general purpose fine-tuning recipe for retrieval-augmented generation, combining pre-trained parametric memory and non-parametric memory.

It makes it possible to add knowledge to the models, thanks to external data sources, documentation, examples, and more. This enables LLMs to handle topics in a more accurate and specialized way, even if their original training did not have sufficient knowledge of those topics. Traditional LLM models cannot easily expand or revise their memory, do not provide a direct analysis of their predictions, and may produce hallucinations. Hybrid models, which combine parametric and nonparametric (retrieval-based) memory, can address these problems because knowledge can be directly revised and expanded, and accessed knowledge can be inspected and interpreted [14].

For example, an LLM that has not been specifically trained on embedded systems or IoT protocols can still help effectively with technical tasks, such as configuring microcontrollers or debugging firmware, by leveraging curated external content through the retrieval mechanism. This capability significantly expands the applicability of LLMs to domains that require precision, real-time updates, and domain adaptation.

This makes RAG particularly useful in scenarios such as configuring peripherals on microcontrollers, interpreting error messages, or adapting code for constrained environments. Suppose that a developer asks *How do I configure an I2C peripheral on an ESP32?*, he would receive an accurate and context-aware response based on up-to-date documentation.

2.4.2 General architecture of RAG systems

To retrieve information, it follows these fundamental steps, as can also be seen in Figure 2.7:

- **User query**: the initial part includes a request in natural language, like *"How can I implement bluetooth on a ESP32 board?"*
- **Retrieval phase**: the query is transformed into a semantic embedding and passed to a retriever, which searches through a vector index populated with relevant documents. This index may contain excerpts from data sheets, ESP-IDF documentation,

code examples, and annotated tutorials. The retriever returns the top-k most semantically relevant text chunks, where k is a predefined number.

The retrieval component is based on a bi-encoder architecture, such as the Dense Passage Retriever (DPR). This system produces a dense representation of both the query and the documents. Finding the top-$k$documents with the highest prior probability is a Maximum Inner Product Search (MIPS) problem, which can be solved in approximately sublinear time [14].

The index is often built using embedding models, in istance OpenAI, Azure OpenAI, or SentenceTransformers, and stored in a vector database like Azure Cognitive Search, FAISS, or Weaviate. The document index is also called non-parametric memory.

In the context of this work, the **SearchClient** class from the *Azure.Search.Documents* library was used to query Azure Cognitive Search indexes and retrieve relevant documents efficiently. This client abstracts the complexity of the full text search and provides semantic ranking and filtering capabilities [18].

- **Generation phase**: the retrieved documents are injected into the prompt as context, typically using a predefined template. The LLM then generates a response that is not only syntactically coherent but also grounded in the retrieved content. This helps avoid hallucinations and improves factual consistency. The model can draw on its intrinsic parametric knowledge or limit responses to the information provided, and can integrate conversation history for multi-turn interactions [8].

  The generator component can be modeled using any encoder-decoder. A common example is BART-large, a pre-trained seq2seq transformer [14]. The retrieved content and the input are concatenated for processing by the generator.

  The output could be different; it depends on the documents and data uploaded in cloud, some examples of the output could be the implementation the requested function, a text that specifies how to implement something, a text that explains what was asked. RAG models tend to generate more specific, more varied, and more factual language than a purely parametric seq2seq baseline.

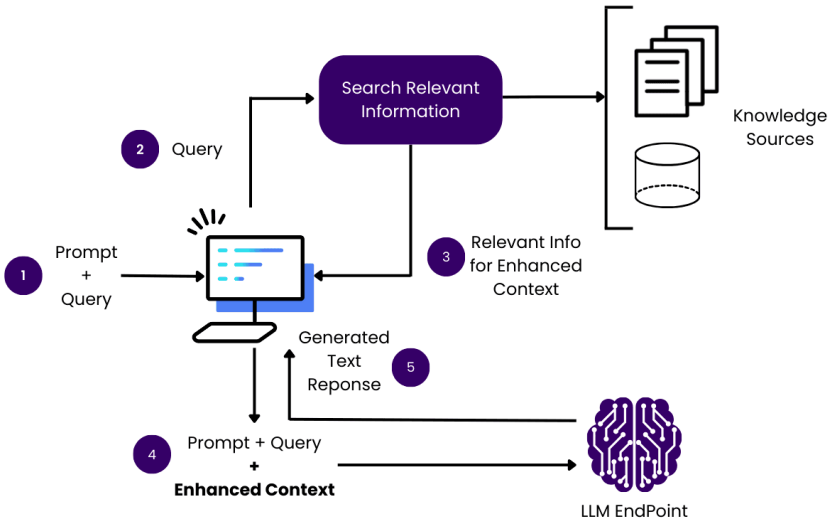

Figure 2.7: Standard flow of a RAG system. Source: [1]

2.4.3 Integration with Azure and the SearchClient class

In this thesis, Azure Cognitive Search was used to implement the retrieval module, integrated by the *SearchClient* class of the Azure.Search.Documents package [18]. The *SearchClient* class from the Azure.Search.Documents library was used to query the Azure Cognitive Search indexes and retrieve relevant documents efficiently. This client abstracts the complexity of the full-text search and provides semantic ranking and filtering capabilities.

SearchClient provides a high-level interface to:

- Create semantic queries on indexed content.
- Retrieve relevant documents based on vector similarity or semantic ranking.
- interface with indexes created from heterogeneous sources, such as technical documentation, code, markdown files, or PDF files.

This component has proven to be fundamental to enrich the LLM generation with updatable and specific content for the IoT and embedded domain, without the need for fine-tuning the model.

2.4.4 Advantages of the RAG paradigm

The Retrieval-Augmented Generation (RAG) paradigm offers an amount of significant benefits that distinguish it from Large Language Models (LLM), which are purely parametric, overcoming many of their intrinsic limitations. Some of these are:

- **Upgradability**: it is one of the most important benefits, documents can be updated or replaced without the need to retrain the model, allowing the model's knowledge to be updated simply by replacing its non-parametric memory.
- **Domain-adaptation**: RAG models enables the integration of highly specialized knowledge (e.g. datasheets, tutorials, application notes).
- **Transparency**: it is possible to show the user the documents used in the generation, improving traceability and providing a form of interpretability to the model, since the memory is human-readable. This contrasts with purely parametric models that are often considered like black boxes.
- **Improved Factuality and Specificity**: they generate more factual and specific responses than purely parametric models, significantly reducing hallucinations [10].
- **Superior Performance on Knowledge-Intensive Tasks**: RAG models achieve state-of-the-art results in a wide range of knowledge-intensive Natural Language Processing (NLP) tasks, such as answering open domain questions [14]. They can generate correct answers even when the answer is not verbatim contained in the retrieved documents.
- **Parameter Efficiency**: RAG models can achieve high performance with significantly fewer trainable parameters than large purely parametric models.
- **Dynamic and Flexible**: RAG can result very useful for dynamic environments that require real-time updates, unlike fine-tuning that requires retraining for each update.

2.4.5 Challenges and limitations

Despite the benefits, the RAG approach has some critical problems and limitations that must be carefully considered for its successful implementation.

- **The quality of the response strongly depends on the quality and coverage of the indexed corpus**. The retrieval phase may suffer from low precision and recall, leading to the selection of misaligned or irrelevant blocks and the loss of crucial information.
- **Retrievers must be carefully configured** (filtering, ranking, vector search). Pre-retrieval strategies, like indexing optimization, query rewriting or expansion, and post-retrieval strategies, such as context reordering or compression, are needed to improve retrieval accuracy.
- The integration between the retriever and the generator requires being careful in managing the prompt to **avoid context overflow**. Excessively long contexts can lead to the *Lost in the middle* problem [8], where the LLM focuses only on the beginning and end of the text, forgetting the intermediate information. That integration can also result in outputs that are uncoordinated, inconsistent, or redundant.
- In some preliminary experiments, a **retrieval collapse** phenomenon was observed for some tasks, where the retrieval component learned to retrieve the same documents regardless of input. In these cases, the generator learned to ignore the documents, and the RAG model performed equivalently to a purely generative model [8].
- Generation models may **overly rely on augmented information**, simply repeating retrieved content without adding summary or insight.
- The potentially **presence of noise or contradictory information** during retrieval can negatively affect the quality of the RAG output.
- RAG, although it is more efficient in terms of dynamic knowledge than pure LLMs with long contexts, could incur **higher latency**, given by the retrieval phase.

Chapter 3

Materials and methods

3.1 Introduction

This chapter illustrates the practical steps and architectural decisions taken during the development of the system.

3.1.1 Single-agent system

The development started with a single-agent architecture where a language model, enhanced with retrieval capabilities, could generate C code in response to natural language tasks. This monolithic approach has been demonstrated to be effective for initial experiments, but has revealed limitations in terms of modularity, task specialization, and conversational flexibility. For this reason, the system was later redesigned as a multi-agent architecture, introducing collaborative agents with specialized roles such as task decomposition, code validation, and iterative refinement. It also describes the methodology followed, the integration of Retrieval-Augmented Generation (RAG), detailing the agent design, their interaction flows, and the overall development pipeline used to prototype and validate the system.

3.1.2 Multi-agent systems

Initially, the architecture was centered around a single, general-purpose agent capable of interpreting natural language requests and producing C code accordingly. While effective for constrained use cases, this setup lacked support for more complex scenarios, such as iterative validation, targeted file modifications, or coordinated subtasks. The single agent only generated code snippets within the chat.

Because of this, the system was restructured into a modular multi-agent environment, inspired by principles of collaborative software development. Each agent was designed with a well-defined responsibility, such as project planning, code building, bug fixing, reviewing, and final approval. This separation of concerns improved clarity, reusability, and adaptability, while enabling richer conversational interactions and improved handling of state across multiple turns.

The multi-agent architecture for generation tasks leveraged Retrieval-Augmented Generation (RAG) as well, to enhance the contextual awareness of the language model. RAG plays a key role in grounding the generative outputs in existing ESP-IDF documentation and code examples. By retrieving relevant information from a structured index of ESP-IDF projects, the model could answer more precisely and generate code consistent with

library conventions. In the multi-agent system, RAG is used by several agents, particularly during code generation. For example, the agent responsible for project generation can leverage retrieved examples given by another agent. This consistent access to external knowledge sources significantly improved both the accuracy and robustness of the overall system.

The system relies on *Azure OpenAI* infrastructure to power its LLM agents, and on *Azure Cognitive Search* to provide retrieval capabilities. The models used for code understanding and generation are based on GPT-4, specifically the `gpt-4o` variant deployed through Azure. Each agent operates on a shared API configuration, ensuring consistent performance and centralized control over resource usage.

For the retrieval component, *Azure AI Search* [18] was used with a custom-built index containing structured code snippets and documentation. This allows the system to retrieve semantically relevant code samples.

3.1.3 Reasons for editing separation: generated and existing projects

While designing the multi-agent architecture for modifying ESP-IDF projects, it was considered appropriate to clearly distinguish the operational flow and the behavior of the agents in two distinct cases, projects generated by the system and existing projects loaded by the user. The separation between these two contexts is not only a matter of logical organization but is motivated by deep architectural, operational, and scalability differences that significantly impact the behavior of the system.

In the case of a generated project, the entire content of the source code has been produced within the system itself. This implies that the structure, file naming, module logic, and internal organization of the project are known, controlled, and compliant with a predefined standard. As a result, agents can operate with complete and reliable knowledge of the project context, and each modification activity is configured as a coherent extension or refinement of an already valid and synthetic base.

Alternatively, an existing project may have a complex, extensive, undocumented, or non-standard structure. Often, these projects include hundreds of files, unknown dependencies, and idiosyncratic configurations that make it difficult for agents to have a complete view of the code. The loading phase alone may be burdensome or inapplicable, either due to contextual memory constraints (token limits) or for performance or readability reasons.

For generated projects, it is possible to directly load the entire code base into the context of the conversation, due to its small size and predictable structure. Unlike existing projects, it was necessary to design a separate flow, where the system does not load the entire project into memory, but adopts an incremental and selective code exploration mechanism. In practice, agents analyze the project dynamically and identify files potentially relevant to the user's change request. Only those files are loaded and processed. This approach ensures scalability and stability even with very large projects but implies a completely different management logic, which would not be applicable or useful in cases where the project has just been generated by the system.

The types of changes requested by the user in the two cases are also different. In generated projects, changes are mainly configured as additions or customizations guided by the initial prompt, which can be treated as an extension of the project itself. In existing projects, however, changes can concern local refactoring, integration with existing modules, or correction of legacy code, which require a more cautious, iterative, and contextualized logic, often constrained by compatibility with existing code of which the system does not have full knowledge.

For these reasons, the design envisages two separate architectural flows: one dedicated to editing generated projects, where the entire context of the project is known and can be treated globally; and one dedicated to editing existing projects, where it was necessary to develop a distinct logic, based on targeted exploration and incremental management of the context. This separation ensures greater efficiency and clarity in the implementation, and also allows the system to adapt more robustly to real use cases, maintaining high quality of the code produced and suggestions generated.

3.2 Initial Prototype: Single-Agent System

The first version of the system was based on a single-agent architecture capable of taking a user query in natural language and generating executable C code for ESP-IDF platforms. The agent leverages a Retrieval-Augmented Generation (RAG) setup to increase accuracy and domain specificity.

Upon receiving a user task, like *Configure an I2C sensor on ESP32*, the agent performs a semantic search over a vectorized index containing ESP-IDF code examples and documentation. The top-k most relevant documents are retrieved and injected into the prompt context sent to the language model. The model then synthesizes the final code snippet based on these examples, adhering to the syntactic and semantic conventions of the ESP-IDF framework.

Although this initial prototype demonstrated the core concept, such as the ability to generate contextually appropriate code based on real ESP-IDF examples, it had several intrinsic limitations. The system's output was confined to a textual response printed directly to the console. At this stage, no additional tooling, project generation, or interactive user experience was integrated. Consequently, the result of the generative process remained detached from any form of structured software development pipeline, which is especially important in embedded environments where file organization, configuration files, and hardware-specific integration play a central role.

In this regard, the single-agent approach proved to be too limited to satisfy the broader objective of the project, namely, to build a system that could not only generate isolated code fragments, but also support the iterative construction of complete functioning embedded applications.

These observations motivated a substantial architectural evolution of the system. Specifically, the single-agent model was extended into a multi-agent conversational architecture, where multiple specialized agents could collaborate dynamically within a shared dialogue context. This allowed the system to move beyond static code generation, allowing more complex interactions such as planning, task decomposition, validation, and iterative refinement. The transition to a multi-agent design marked a shift from a passive code generation tool to an active, interactive assistant capable of coordinating the development of full embedded projects through structured and collaborative reasoning.

3.2.1 RAG Implementation in single-agent

Since the theoretical foundations of Retrieval-Augmented Generation (RAG) were introduced in Section 2.4, this section focuses on the real implementation of the RAG pipeline within the developed embedded code assistant system. The objective was to consent the language model to access domain-specific technical information, such as ESP-IDF documentation, code examples, and tutorials, without requiring parameter tuning or model retraining. This was achieved through a seamless integration between Azure Cognitive

Search and a prompt-based language model interface.

Architecture and Data Sources

As mentioned in Section 2.4, the retrieval component was implemented using Azure Cognitive Search [18], chosen for its ability to handle both dense vector search and full-text semantic ranking. The indexed corpus included a heterogeneous set of documents, such as:

- ESP-IDF official documentation (scraped and segmented into topic-specific chunks);
- Annotated code examples from official GitHub repositories;
- Markdown tutorials and how-to guides;
- Snippets of frequently used peripheral configuration patterns.

Each document was pre-processed and chunked using a sliding window strategy with a predefined token overlap, to ensure semantic cohesion within each indexed unit. The resulting chunks were embedded using a model compatible with Azure's embedding services and stored in a vector index accessible through the *SearchClient* class from the Azure.Search.Documents SDK.

Query Processing and Retrieval Logic

When a user submitted a query in natural language, the system first transformed the query into a dense vector representation. The *SearchClient* class was then used to perform a semantic search against the Azure index, retrieving the top-k most relevant document chunks. Semantic filters could optionally be applied to limit the search to specific types of content (e.g., only code examples or textual explanations).

The retrieval configuration included both vector similarity search and keyword-based fallbacks, in order to compensate for edge cases where vector-based retrieval alone produced irrelevant results. This hybrid approach helped increase robustness in the presence of ambiguous or poorly phrased user queries.

Prompt Construction and Generation Phase

The retrieved documents were formatted and injected into the language model prompt using a structured template. This template included:

- A clear task description based on the user's query;
- A list of retrieved documents (with optional titles and metadata);
- Instructions to the model to base its response strictly on the provided documents.

An example version of the prompt structure, used for the single-agent, was as follows:

```
You are an expert in ESP-IDF and help developers to create IoT applications. Your Task is
[Document 1]
[Document 2]
...
User question: How do I implement UART on ESP32?
Answer: ...
```

This format helped constrain the model's response to be grounded in the retrieved content, minimizing hallucinations.

Challenges Encountered in Practice

Despite the advantages of RAG, its integration posed several practical challenges. One of the most significant issues was the semantic granularity of the indexed documents. Some chunks were either too narrow *(losing important context)* or too broad *(causing prompt overflow)*. Fine-tuning chunk size and overlap was crucial to ensure retrieval accuracy without exceeding token limits.

Another recurrent issue can be the presence of redundant or noisy documents in the corpus. In some cases, similar examples or outdated tutorials interfered with the generation phase, leading to repetitive or contradictory answers. Filtering and curation of the indexed corpus was, therefore, an ongoing maintenance task.

Additionally, the system can occasionally suffer from retrieval collapse, where identical documents were returned regardless of query variation. This was mitigated by periodically refreshing embeddings and updating document representations to reflect structural improvements in the dataset.

Lastly, managing prompt length limitations became a constraint, especially when attempting to include multiple high-quality documents along with a rich user context. Techniques such as document ranking, summarization, and dynamic prompt truncation were explored to improve efficiency without compromising informativeness.

The integration of RAG into the embedded assistant system proved to be a key enabler for accurate and context-aware code generation. By externalizing domain knowledge into searchable memory, the system achieved a high degree of adaptability and specificity without modifying the language model itself. Although implementation challenges emerged, particularly around corpus quality and prompt length, these were addressed through engineering trade-offs and iterative refinement of the retrieval pipeline.

3.3 Architectural shift: from single to multi-agent system

The initial development of the embedded code assistant was driven by the goal of validating whether a language model, when coupled with a retrieval mechanism, could autonomously generate correct and context-aware code for ESP-IDF platforms. Although the single-agent prototype confirmed the feasibility of this objective in a controlled and narrow setting, it soon became apparent that this architecture was fundamentally insufficient to support more sophisticated, real-world development workflows. This realization marked the beginning of a substantial architectural transformation.

3.3.1 Conceptual limitations of the single-agent architecture

The single-agent paradigm, by its nature, imposed a strict coupling between task interpretation, information retrieval, and code generation. All responsibilities were delegated to a single entity, which initially proved useful for simple tasks. The system lacked the flexibility and modularity required to handle more advanced operations such as project structuring, code verification, or interactive development workflows.

A central issue was the system's lack of specialization. Without a mechanism for delegation or task decomposition, all decisions and computations, whether semantic, syntactic, or procedural, had to be inferred and handled by a single, stateless entity. Although simple, this design was inherently limited in terms of scalability, error recovery, and adaptability.

3.3.2 Constraints encountered in single-agent approach

Lots of practical challenges emerged while testing the system in increasingly complex tasks. The most critical limitations of the single-agent system encountered are the following.

- **Token and Context Limitations**: it is one of the first constraints encountered. The system quickly reached token limits when trying to include retrieved documents, user queries, generation templates, and previous context in a single prompt. Due to this constraint, the model struggled to process long or multi-step instructions and forced frequent truncations which negatively affects the output quality.
- **Lack of File System Integration**: the agent was limited to producing plain text output via the console and exhibited no awareness or interaction with the file system. As a result, it was incapable of autonomously generating essential components such as CMakeLists.txt, sdkconfig, header files, build scripts, or integrating external dependencies.

  Since the aim of the project was to produce a fully functional and executable ESP-IDF application, this lack of integration represented a significant limitation. The inability to define and create the required files and directories restricted the agent's capacity to deliver a complete and buildable project.
- **No Verification or Build Process**: there is no feedback to verify whether the generated code was valid, compilable or executable. This makes it impossible to detect errors or fix the generation process in response to actual build results.
- **Inability to Coordinate Multiple Steps**: tasks that require sequential actions, like generating a peripheral driver, integrating it into a larger project, configuring build options, and validating functionality, could not be managed by an atomic action. Each step had to be handled in a single, often overloaded prompt.
- **No Error Handling or Correction**: in the event of code generation errors, incorrect assumptions, or inconsistent results, the system does not offer a way to detect or correct mistakes autonomously.

These limitations demonstrated that the single-agent architecture, while sufficient for validation and experimentation, was inadequate for realistic embedded development use cases.

3.3.3 Motivation for a Multi-Agent Paradigm

In response to these constraints, the architecture was restructured around a multi-agent model, where discrete agents, each with a specialized function, collaborate to accomplish complex tasks. This transaction was inspired by multi-agent system (MAS) principles, particularly those used in collaborative problem solving and distributed AI.

The multi-agent system introduced the following key advantages.

- **Functional Decomposition**: each agent is assigned a specific role within the system, such as generation, validation, building, and refinement. This division allows for clear separation of concerns, reducing cognitive complexity and improving maintainability. By encapsulating each function in an independent agent, the system can evolve each component independently. For example, improvements to the build-checking logic can be made without altering the generation or validation agents.

This decomposition also allows for prompt focus on design and evaluation strategies related to the agent's specific objective.

- **Scalability and Modularity**: agent-based architecture is inherently extensible. New tasks or capabilities can be added by defining new agents rather than modifying existing ones. For instance, to introduce a dependency resolution capability or an agent specialized in documentation generation, it is sufficient to define their logic and plug them into the coordination flow. This modularity accelerates development and also allows for experimentation and incremental improvement, since agents can be tested alone or deployed conditionally based on task complexity or project requirements.

- **Contextual Coordination**: agents work within a shared conversational context, either within the same chat instance or across coordinated chats. This enables implicit communication through message history and explicit handover of intermediate outputs. For example, a generation agent can produce an initial draft of a source file, which is then passed to a validation agent for semantic checks and finally to a build agent for compilation testing. This cooperative reasoning process simulates a real collaborative development setting, improving the alignment between steps and allowing agents to build on each other's contributions rather than starting from scratch.

- **Pipeline Automation**: the system is capable of automating complex multi-step development workflows. Tasks like initializing a project structure, generating configuration files, invoking build commands, analyzing errors, and refining the source code are performed in sequence by specialized agents. This enables end-to-end task execution from high-level intent to a correct working project, without manual intervention at every stage. This structure supports progressive execution and correction, increasing the robustness and the task success rate.

- **Resilience and Redundancy**: The introduction of feedback mechanisms between agents improves fault tolerance. If a build fails or the validation agent detects incorrect logic, the system can dynamically trigger corrective actions. This redundancy avoids task failure in the presence of partial errors and mimics the iterative trial-and-error process of human developers. Furthermore, agents can be reused as fallback mechanisms.

- **Enhanced Interpretability**: Multi-agent workflows produce traceable and auditable logs of the entire development process. Each input, output, and reasoning steps of the agent are recorded in the conversational flow, leaving users and developers to inspect the rationale behind decisions or investigate sources of error. This transparency is especially valuable in debugging scenarios or when fine-tuning agent behavior. Knowing which agent did what, the system becomes more intelligible and trustworthy, both during development and when deployed in real-world use.

This architectural evolution reflects a conceptual change from a monolithic to a collaborative system capable of reasoning, planning, and refining outputs across multiple domains of expertise. It more closely resembles a real software development team, where each member contributes their expertise to solve parts of a bigger problem.The assistant is no longer a passive text generator but an active coordinator of tasks with internal dialogue and memory.

The move to a multi-agent system therefore marked a turning point in the development of the project. It enabled the realization of its true objective: not just to generate code, but to orchestrate the creation or modification of complete, valid, and buildable embedded software systems in a conversational, user-centered workflow.

3.4 Design and implementation of multi-agent chats

The transition from a single-agent to a multi-agent architecture required a structural redesign of the entire conversational flow. Initially, the development was guided by examples provided by the official AutoGen documentation [15], which demonstrate the interaction between autonomous agents in coordinated environments to solve complex tasks. Following this approach, a first version of a multi-agent system was created, which is capable of generating ESP-IDF projects for ESP32 board starting from a user request in natural language, with the possibility for each agent to perform well-defined tasks and coordinate through a shared conversational channel. In this implementation, the RAG method is applied in one agent that biases other agents.

3.4.1 Initial multi-agent design for project generation

General architecture

The system is based on *SelectorGroupChat*, a construct provided by AutoGen that allows a group of agents to interact and autonomously select who should act in the next round. The group consists of five agents: *PlanningAgent*, *CodeExampleRetrieverAgent*, *CodeGeneratorAgent*, *CodeReviewerAgent*, *FileOrganizerAgent*

The user task is assigned to the *PlanningAgent*, which is responsible for breaking down the request into subtasks, each of which is then delegated to specialized agents, following a structured plan.

Agents

- **PlanningAgent**: this agent acts like a project manager. It receives the user's task, an example should be *Generate an I2C driver for ESP32*, and splits it into specific subtasks, each for another agent. The planning is textual, with a list structure, for example, `1. CodeExampleRetrieverAgent: Search for I2C driver examples`. The planner does not execute the tasks, but simply coordinates the chat.
- **FileOrganizerAgent**: this agent is designed to take care of the organization of files. Based on extension and content, files are sorted into different directories, like `main/`, `main/include/`, `main/src/`, or `main/misc/`. Its objective is to specify the file names with a description that tells the purpose of each file. An example of its output that can be written in chat is:

```
[
    {
        "file_name": "CMakeLists.txt",
        "description": "CMake configuration for the overall project."
    },
    {
        "file_name": "main/CMakeLists.txt",
        "description": "CMake configuration for the project main directory."
```

```
    },
    {
        "file_name": "main/main.c",
        "description": "Entry point of the application."
    },
    {
        "file_name": "main/src/wifi.c",
        "description": "Implementation of WiFi connection management."
    },
    {
        "file_name": "main/inc/wifi.h",
        "description": "Header file for WiFi connection management."
    }
]
```

- **CodeExampleRetrieverAgent**: specialized in querying a semantic index on Azure AI Search, this agent has exclusive access to the `retrieve_docs` tool, which allows to get relevant code fragments from the ESP-IDF corpus. Its behavior is strictly limited to semantic querying, so the agent from the given user task and the project structure creates an input query to retrieve potentially useful code, implementing the RAG technique. The results are sent in chat and used by other agents for generation.
- **CodeGeneratorAgent**: this is the agent responsible for generating ESP-IDF code. It produces JSON objects with the **file_name** and code fields, according to the instructions received from the planning agent, the organizer file agent, and the references returned by the retriever. It only generates all the project files, it does not save them directly, but gives the content to the reviewer for a preliminary evaluation.
- **CodeReviewerAgent**: the presence of this agent has been shown to be crucial to improving the robustness of the system. During the first iterations, it was common for the generated code to have syntactic errors or incompleteness. The reviewer is responsible for checking the correctness of the generated files and, if suitable, triggers saving by the `generate_code` tool. Otherwise, it provides explicit feedback to the generator agent so that it can recreate the incorrect file code.

Coordination and selection of agents

To coordinate the flow of the chat, the system uses a dynamic selector prompt, which provides participants with the conversational context and available roles. Only one agent is active at a time and is automatically selected according to the instruction given by the planner agent, conversation history, and priority rules; the planner must always be the first to act. The chat will close when the planner agent writes the word *TERMINATE*, that is when all files are saved successfully.

In Figure 3.1 there is a representation of the chat workflow, the green arrows represent the flow coordinated by the planner agent, the dashed arrow is used to specify that the information sent is reused by that agent, and the purple boxes indicate the tools of an agent. As said, the planner agent has only the role of coordination, so it iteratively says which agent should act next. The organizer starts to write the project structure, then the retriever agent retrieves the code from the document querying in *Search Client* (Section 2.4.3). Now, it is the generator agent's turn that writes all file, afterwards the

reviewer review the generated file, it saves the correct files, if possible correct the code and save the file or ask the generator agent to regenerate the file code.

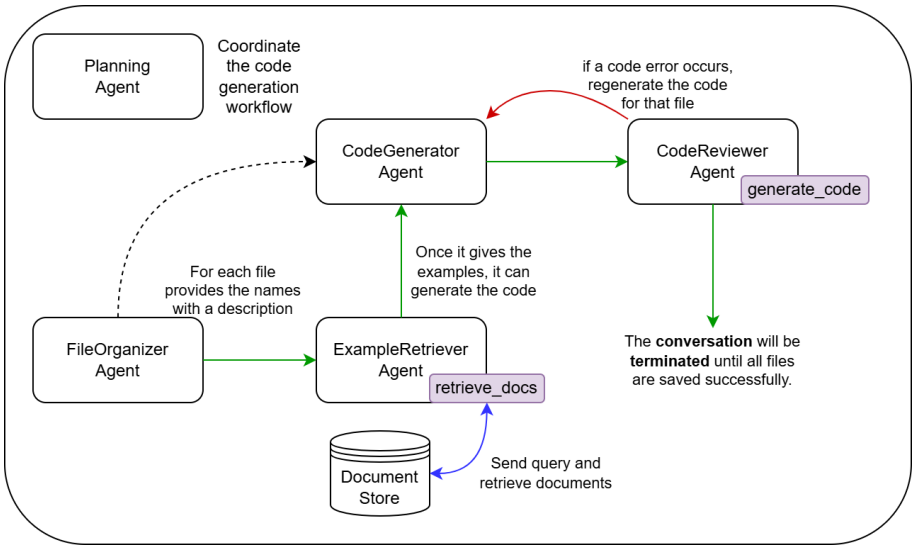

Figure 3.1: Initial multi-agent chat workflow for generation

This first multi-agent implementation represented a significant step forward compared to the single-agent system. The generated design is structurally correct, and the interaction between agents allows for a better division of the work, as well as introducing a first level of automatic verification. However, there is no guarantee that the code will work as expected or that it will be compilable, unless a manual intervention is performed, such as running the build by the user.

These limitations motivated the design of more complex tasks, which are described in the following sections.

3.4.2 Multi-agent design for project generation tasks

This section describes the final architecture implemented for the ESP-IDF design generation tasks, which represents a substantial evolution over both the initial single-agent prototype and the multi-agent design discussed in the previous section. Although those earlier systems demonstrated the feasibility of decomposing generation tasks and coordinating multiple agents, they lacked robustness in ensuring that the resulting codebase was fully functional, compilable, and aligned with the constraints of real-world embedded development workflows.

The primary goal of this redesign was to close the semantic gap between the code that might be correct and the code that actually compiles and runs on a target ESP32 microcontroller. This meant not only improving internal coordination between agents but also enhancing the reliability of the output through external validation steps. In particular, special attention was paid to reducing silent failure modes and promoting transparency and accountability throughout the generation process.

For architectural reasons, especially those related to context window limitations and token overflows in large-scale project generation, the overall pipeline was split into two distinct but interlinked phases.

- **Phase 1. Code generation**
  This phase focuses on the creation of all project files. Architecturally, it is very

similar to multi-agent pattern as the prototype described in Section 3.4.1, with improvements aimed at enhancing file naming conventions, termination handling, robustness against malformed JSON structures, and adjusting agent prompts using advanced prompt engineering techniques [3].

- **Phase 2: Build and fix loop**
  Once the project is generated, it undergoes a build process. This phase introduces a critical verification step: the output of the build process determines whether the chat concludes successfully or a correction cycle is triggered. If the build fails, the system extracts the error messages and engages in an iterative correction loop, where agents attempt to fix the code until either:
  - the build succeeds
  - a predefined maximum number of fixing attempts is reached.

This two-phase design enables a more realistic and production-oriented development pipeline, effectively bridging the gap between code generation and actual deployment on hardware targets. By separating the generation and validation steps, the system not only produces syntactically correct code, but also verifies its correctness through a real build attempt. This approach introduces a feedback loop that allows one to automatically detect and correct compilation errors.

First chat - generation phase

General architecture

The architecture of the code generation phase reuses the same foundational structure based on *SelectorGroupChat*, which orchestrates the interaction of multiple agents and determines, at each conversational turn, the most appropriate agent to act based on context and task progression.

Despite its structural similarity to the prior implementation, this phase incorporates substantial improvements. The team of agents is more clearly defined, each with highly specialized roles and enriched with detailed and purpose-built prompts that better constrain their behavior and guide their contributions [3]. Furthermore, the approval agent and a tool for the file organizer agent have been added.

Agents

- **PlanningAgent**: this agent acts like a project manager. It receives the user's task, an example should be *Generate an I2C driver for ESP32*, and splits it into specific subtasks, each for another agent. The planning is textual, with a list structure, for example, `1. CodeExampleRetrieverAgent: Search for I2C driver examples`. The planner does not execute the tasks, but simply coordinates the chat.
- **FileOrganizerAgent**: this agent is designed to take care of the organization of files. Based on extension and content, the files are sorted into different directories, like `main/`, `main/include/`, `main/src/`, or `main/misc/`. Its objective is to specify the file names with a description that tells the purpose of each file.

  Before producing its final output, the agent invokes the `file_name_checker` tool, which receives the list of proposed filenames and verifies whether any of them are blacklisted. If one or more names are flagged (e.g., due to conflicts with standard

ESP-IDF libraries), the agent automatically renames those files to avoid collisions. This validation step is critical to avoid errors during compilation. For example, naming a user-defined header file `mqtt_client.h` could cause conflicts with the native ESP-IDF library of the same name when included by `#include "mqtt_client.h"`. In such a case, the compiler might inadvertently reference the local file rather than the intended system header, leading to hard-to-debug errors. The renaming mechanism removes these issues by proactively enforcing a secure and unique namespace for agent-defined files.

- **CodeExampleRetrieverAgent**: specialized in querying a semantic index on Azure AI Search, this agent has exclusive access to the `retrieve_docs` tool, which allows to get relevant code fragments from the ESP-IDF corpus, implementing the RAG technique. Its behavior is strictly limited to semantic querying, so the agent from the given user task and the project structure creates an input query to retrieve potentially useful code. The results are sent in chat and used by other agents for generation.
- **CodeGeneratorAgent**: this is the agent responsible for generating ESP-IDF code. It produces JSON objects with the **file_name** and code fields, according to the instructions received from the planning agent, the organizer file agent, and the references returned by the retriever. It only generates all the project files, it does not save them directly, but gives the content to the reviewer for a preliminary evaluation.
- **CodeReviewerAgent**: the reviewer is responsible for checking the correctness of the generated files and, if suitable, triggers saving by the `generate_code` tool. Otherwise, it provides explicit feedback to the generator agent so that it can recreate the incorrect file code.
- **ApproveAgent**: this agent acts as a final gatekeeper, ensuring that the entire code generation pipeline is completed in a controlled way. Its main purpose is to authorize the termination of the conversation only after all generated files have been correctly processed and stored. It has to call the tool `approve_saved_files` to close the conversation.

It is good to point out that in previous implementations, the conversation could be terminated inadvertently as soon as the keyword `TERMINATE` appeared in the chat, typically triggered by the *PlanningAgent* after completing its task delegation. However, this approach proved to be fragile, as termination could occur even if some file outputs had not yet been validated or saved, potentially leading to incomplete or corrupted projects.

To address this issue, the *ApproveAgent* was introduced as a dedicated validation layer. That agent waits until the file names given by *FileOrganizerAgent* have been successfully saved, and only then evaluates if the chat can be safely ended. Only after this verification step does the agent emit the termination signal, ensuring a clean and complete closure of the workflow.

This mechanism adds robustness and safeguards the pipeline against premature termination, reducing the risk of incomplete outputs and improving overall reliability, especially in longer multi-step generations where asynchronous agent interactions can introduce inconsistencies.

Coordination and selection of agents

The multi-agent system relies on a dynamic coordination mechanism managed through

a *SelectorGroupChat*, which ensures that only one agent speaks at a time, based on the current state of the conversation. Coordination is primarily guided by the instructions of the *PlanningAgent*, who acts as a central dispatcher and defines the expected sequence of operations through structured task delegation. This approach ensures that agents operate within their designated scope, reducing redundancy and minimizing conflicting actions.

The agent selection logic leverages a dedicated selector prompt, which includes contextual information such as conversation history, available tools, and each agent's declared role. The planner is always the first to act, as it provides the overall decomposition of the task. Subsequent activations follow a logical progression shaped by the planner's delegation and the evolving conversational context.

Each agent responds autonomously when selected, using tools if needed, and interacting only within its responsibility domain. This tightly coupled coordination scheme allows the workflow to proceed in a disciplined and deterministic manner, without requiring global awareness or direct communication between agents.

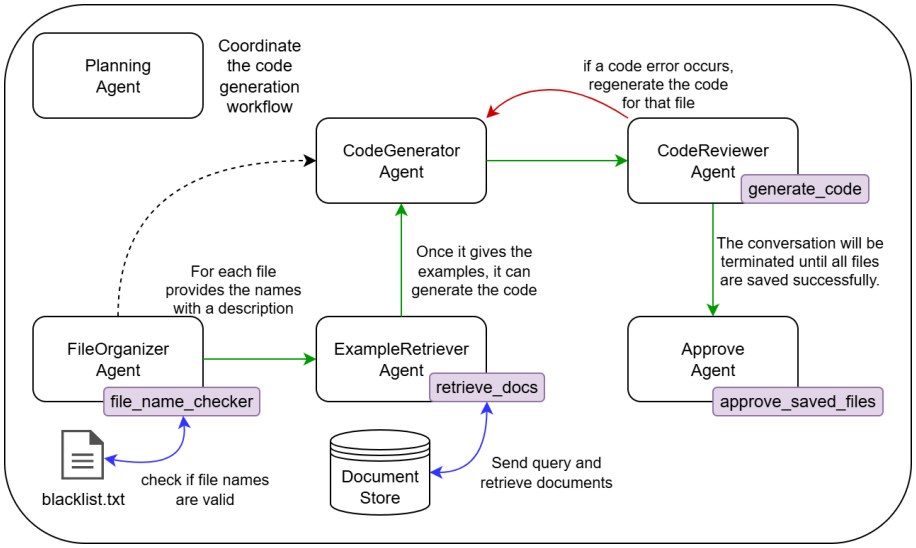

Figure 3.2: Multi-agent chat workflow for project generation

The execution flow is further enhanced through the explicit use of tools, which are invoked using function calls. In the diagram shown in Figure 3.2, green arrows represent the sequential speaking turns among the agents, reflecting the order in which they participate in the conversation. The blue arrows illustrate the connection between external data and tools; when there are problems on files that should be saved, it takes a step back to regenerate it. Purple boxes denote tool invocations, illustrating key moments in which agents interact with the external environment to validate or retrieve information.

The workflow starts with the *PlanningAgent*, which orchestrates the generation process by decomposing the high-level objective into a sequence of well-defined subtasks. These tasks are then delegated to domain-specific agents in a controlled and linear fashion. The first action involves calling the *FileOrganizerAgent*, which defines the target structure of the ESP-IDF project, specifying which files must be created and where they belong within the directory hierarchy. Once the expected file structure is defined, the workflow continues with the *CodeExampleRetrieverAgent*, responsible for searching and retrieving relevant ESP-IDF examples from a dedicated Azure AI-powered index. These examples serve as references to ensure consistency with best practices and compatibility with the targeted SDK version. The retrieved examples and structural requirements are then passed

to the *CodeGeneratorAgent*, which is in charge of producing full implementation files, it generates both .c and .h files. Before any generated code is finalized, it is handed off to the *CodeReviewerAgent*. This agent carefully validates syntax correctness and structural consistency, applying fixes where necessary. Once the review is completed, this agent is the only one allowed to write the finalized files to the project. After all files have been reviewed and saved correctly, the process concludes by calling the *ApproveAgent*, which confirms that the workflow is over.

This workflow design supports a robust and interpretable generation pipeline. The combination of clear agent roles, enforced turn-taking, and dynamic tool invocation allows the system to handle complex project generation tasks in a modular and traceable fashion.

Second chat - building and fixing phase

General architecture

Even in this phase, the interaction between agents is orchestrated through the *SelectorGroupChat* construct, which regulates the speaking turn and dynamically selects which agent should act, based on the state of the conversation and the progress of the task. Nevertheless, compared to the project generation phase, the chat here is optimized for an iterative and controlled refactoring cycle of existing code, starting from a user-uploaded project.

The flow is designed to perform semantic refactoring, for example, to make structural improvements to the code without modifying its behavior. This process consists of a series of cycles made up of generation, validation, and automated verification, supervised by agents specialized in quality control and consistency with the design context.

A distinctive element of this phase is the retry management mechanism, governed by the `max_attempts` parameter, set to a maximum of 10, it can be modified. In case of build process failure, the code will be fixed taking into account the given compilation errors, and when closed a new chat will be restarted until a successfully build process or the maximum number of attempts is reached.

The multi-chat approach keeps each attempt isolated and traceable, making debugging easier and improving the reliability of the entire refactoring cycle. In addition, the maximum retry constraint protects the system from infinite loops or recursive regressions in the presence of persistent errors that are difficult to automatically resolve.

Agents

- **PlanningAgent**: this agant acts as the main coordinator of the refactoring and verification workflow of the ESP-IDF project. Its task is to break down the complex task into an ordered sequence of specific subtasks that are delegated to other specialized agents present in the chat. It does not directly perform operational activities on the code, but defines the execution logic and guides the entire process. *PlanningAgent* is configured with detailed knowledge of the roles of other agents in the system and sets the work strategy according to a well-defined flow. All task assignments follow a structured syntax, in the form `<Agent> : <task>`, making communication between agents transparent and easily understandable.

- **UploadProjectAgent**: it has the task of uploading the ESP-IDF project files into the chat, making them available to other agents for subsequent analysis, correction and validation phases. This agent uses the `read_code_files` tool, a function that

explores the contents of the project's output directory, explicitly excluding directories and files that are not relevant for analysis, such as the `build/` directory, and reads the contents of the project's source files. The files are returned as a concatenation of text blocks containing file paths and contents. This format ensures a clear and segmented representation of the entire project, which can be easily processed by subsequent agents.

Once the agent has uploaded all readable files, its task is considered concluded. In case of errors like non-existent directory or non-decodeable files, the system is able to return appropriate diagnostic messages to facilitate debugging. The *UploadProjectAgent* does not make autonomous decisions and does not actively participate in the correction cycle, but it is essential to correctly initialize the project context, providing a global and readable view of the code base that will be the starting point for the subsequent phases of the process.

- **BuildAgent**: the agent is responsible of the compilation phase of the ESP-IDF project, whose main goal is to verify the structural and syntactic correctness of the code generated or modified in the pipeline.

  This agent has the `build_project` tool, which runs the `idf.py build` command within the correctly initialized ESP-IDF environment. When called, the agent builds the project in an isolated environment. At the end of the process, if the build was successful, the agent sends a confirmation message that allows a direct transition to the *ApproveAgent* closing the conversation. Otherwise, it returns the error log, `stderr`, to assist the *CodeFixerAgent* in diagnosing and resolving the part that triggered the error.

- **CodeFixerAgent**: it is responsible for modifying the code when the build attempt fails. Given the project uploaded by the *UploadProjectAgent* and the feedback produced by the *BuildAgent* performs targeted corrections to resolve the errors. Corrections are made in compliance with ESP-IDF v4.4.8, following the conventions of its folder and file structure. Additionally, the agent ensures proper code formatting, commenting, and the presence of required definitions.

  The output is returned as a structured JSON list of modified source and header files. These modifications are not directly committed to the final project state, instead, they are sent in the chat and then the *CodeReviewerAgent* is in charge of validating the changes before saving them.

- **CodeReviewerAgent**: this agent is the final checkpoint in the code correction cycle. Its role is to perform a thorough validation of the code modifications proposed by the *CodeFixerAgent* before committing them to disk. If the code is valid, the agent saves the files using the `generate_code` tool without any modification. If issues are detected, the agent applies the necessary fixes autonomously and then proceeds with saving. Each file is saved using the original relative path structure to ensure correct integration with the rest of the project. Once all files are successfully validated and saved, the agent notifies the *ApproveAgent* to close the conversation, signaling that the build pipeline is complete.

- **ApproveAgent**: it is used to officially close the conversation. Its activation happens only when all the modified files have been correctly reviewed and saved by the *CodeReviewerAgent*, or when a successful build confirms that no changes are required. The responsibility of this agent is to acknowledge the successful conclusion of

the development pipeline and terminate the session using the `approve_saved_files` tool. By explicitly closing the conversation, the agent guarantees that no further actions will be performed, thus preventing redundant processing or accidental re-execution of earlier stages.

Coordination and selection of agents

In this chat, the agent selection logic is handled by *SelectorGroupChat*, which ensures that only one agent can speak at a time, based on context and the current task. Additionally, a selector function, `selector_func`, has been implemented in the team, which allows to define a function that is used to decide who should speak. Therefore, a standard selector function has been declared, where alternatively the *PlanningAgent* and one of the other agents speak, these to ensure to follow the workflow more strictly.

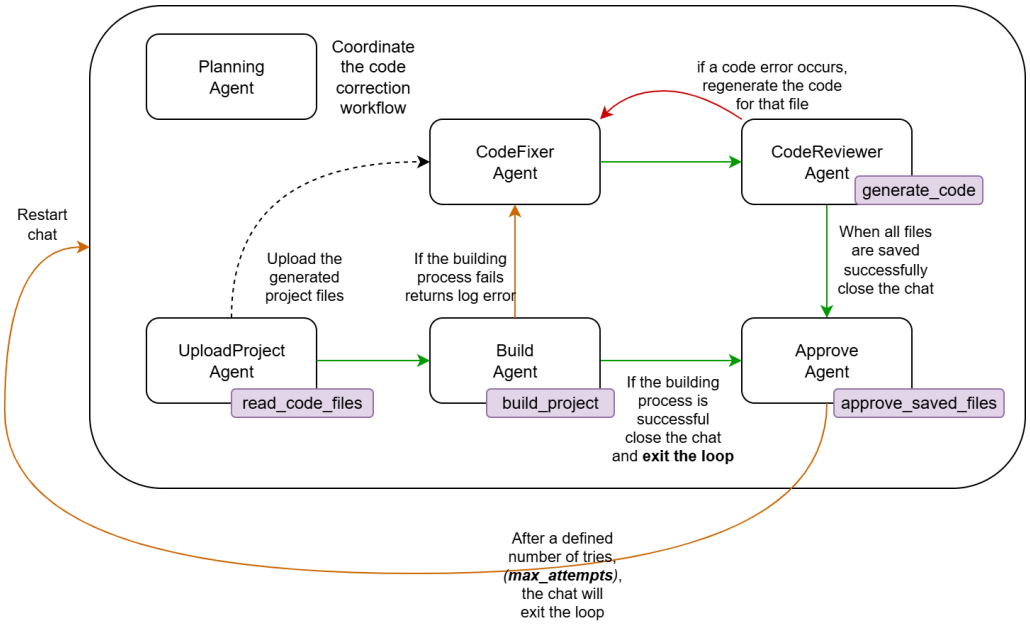

Figure 3.3: Multi-agent chat workflow for iterative fixing project

The schema in Figure 3.3 visually illustrates the second chat flow. This chat iteratively restarts for each failed build process. The green arrows represent the sequential steps between agents, it goes in orange arrows when the build process failed, and in red arrow when there are problems in code that should be saved, while the purple boxes indicate the tools. The flow starts with the *PlanningAgent* which defines the expected sequence of operations through structured task delegation. Then *UploadProjectAgent* is instructed to upload all project files from the output directory. Once all files are available in the chat, the *BuildAgent* compiles the project using the ESP-IDF build system by invoking the `idf.py build` command. If the build completes successfully, the agent immediately signals the end of the session by activating the *ApproveAgent*, which closes the chat. If the build fails, the *CodeFixerAgent* analyzes the compilation errors, the uploaded project files, and proposes corrections. The corrected files are handed to the *CodeReviewerAgent*, which validates the proposed changes. If needed, the reviewer also makes additional corrections before saving the files. If validation fails or files are incomplete, the reviewer requests another intervention from the *CodeFixerAgent*. When all modified files are correctly saved and validated, the *ApproveAgent* close the session, marking the end of the current chat.

Once the conversation is closed by the *ApproveAgent*, the system verifies whether the project build process failed during the previous cycle and whether the maximum number of allowed attempts, defined by the `max_attempts` parameter, has not yet been reached. If both conditions are met, a new chat session is automatically initialized with the fixed code as input, and the entire workflow is restarted from the build phase. This allows for an iterative, traceable, and isolated retry mechanism, where each chat corresponds to a unique fixing attempt. The repetition of the chat ensures that errors are addressed progressively until a successful build is achieved or the retry limit is reached. This design guarantees resilience against initial code instability while avoiding infinite loops in the case of persistent or unresolvable issues.

Each agent is autonomous in making decisions and invoking tools, but turning-taking prevents conflicts and ensures the linearity of the conversation. This design makes the system interpretable and allows you to track, step by step, the path that led to the transformation of the code.

The result is a robust and controllable refactoring pipeline capable of improving code quality while maintaining high operational reliability. The adoption of a threshold on attempts and the presence of a final approver drastically reduce the risk of project corruption or partial changes, and above all allows you to have a compilable project that works on the ESP32 board.

3.4.3 Multi-agent chat for editing task of a generated project

This section shows the final architecture used for the ESP-IDF design editing tasks only for the generated project. When a project is generated, the user can decide to modify something, like adding features, removing characteristics, or modifying functionalities. Initially, this system was meant for all projects, both generated and existing projects, but for architectural reasons they were divided into two different systems. The motivation for the separations of edit tasks has been discussed in Section 3.1.3.

General architecture

The architecture of the generated project modification system is based on the same fundamental structure, *SelectorGroupChat*.This system orchestrates multiple agents, each with a specific role, and dynamically selects the most suitable one to respond at each conversational turn, based on the evolving context and current subtask.

This editing chat assumes that the initial project has already been correctly generated and built, and it focuses on applying incremental changes such as adding new features, modifying existing functionalities, or removing components. Due to the more constrained scope of these interactions, the architecture can maintain a persistent conversational session without frequent resets, unlike the build-and-fix phase, which may require multiple chat restarts to handle token overflow or drastic structural corrections.

Nonetheless, a fix mechanism is still integrated in case new build errors emerge as a result of user edits. This mechanism iteratively analyzes the build error and attempts to apply corrective modifications, similar to the second phase of the project generation pipeline. However, since the initial project is already functional and only minor updates are applied, the probability of having blocking errors, and thus of exceeding token limits or requiring deep recovery steps, is significantly reduced.

Agents

- **PlanningAgent**: it plays a central coordinating role within the multi-agent chat system for editing generated ESP-IDF projects. Its primary function is to act as a task planner and orchestrator, without performing any code modifications directly. Its own objective is to decompose high-level user requests into smaller, well-defined subtasks and delegates them to appropriate specialized agents in the system. This modular approach improves clarity and traceability and ensures that each agent can focus on a specific responsibility.

  The *PlanningAgent* has a knowledge on other agents role and aims. Thanks to the high-level overview and abstracting the orchestration of complex workflows, it ensures modularity, robustness, and maintainability of the editing process within the multi-agent environment. The behavior of this agent strictly adheres to a declarative task assignment format: `<agent> : <task>`

- **UploadProjectAgent**: the agent that is responsible for uploading the contents of the generated project into the multi-agent chat environment, so the other agents can analyze, organize, and modify the project effectively. Its role is fundamental at the beginning of the editing workflow as it allows all relevant source files to be available in the shared conversational context.

  To perform its task, it uses a dedicated tool `read_code_files`. This function performs a recursive scan of the project directory. It systematically excludes build artifacts and auxiliary directories, such as build directory, as well as non-editable or binary-specific files, that are created by the building process. The tool loads the content of each file using this convention:

  ```
  ### File: <relative_path> ###
  <file content>
  ```

- **UnderstandProjectAgent**: it is in charge of performing an analytical pass over the uploaded project files in order to determine the current implementation status and identify any components or functionalities that are missing with respect to the user's editing request. This agent plays a critical role in integrating the gap between the existing project state and the intended modifications, leading the rest of the workflow to operate on precise contextual information.

  Unlike agents dedicated to file operations or structural organization, the *UnderstandProjectAgent* is focused purely on semantic analysis and functional coverage. It neither manipulates the file system nor defines the directory structure, but rather produces a diagnostic report that guides subsequent agents.

- **FileOrganizerAgent**: it plays a pivotal role in structuring and organizing the ESP-IDF project after the analysis phase performed by the *UnderstandProjectAgent*. It does not generate or edit code directly, this agent is responsible for defining the architectural layout of the project, making the decision of which files should exist, where they should be placed, and what purpose they serve.

  In addition, it integrates with a validation tool to ensure the generated file names do not conflict with reserved identifiers, ESP-IDF library names, or disallowed filenames defined in a blacklist. This tool, called `file_name_checker`, verifies the correctness of each proposed filename, it compares file names against a preloaded `blacklist.txt` and flags any that could interfere with core ESP-IDF components or violate naming

constraints. This tool returns either a confirmation that all names are valid, or a list of problematic file names that the agent must be renamed before proceeding.

- **DeleteFilesAgent**: this agent is tasked with maintaining a clean and efficient project workspace by removing unnecessary files, as suggested by the *FileOrganizerAgent*. While the *FileOrganizerAgent* proposes deletions based on project structure and intended functionality, the DeleteFilesAgent has the final responsibility of validating whether these files are truly expendable before executing the deletion, this step is made to avoid accidental removal of files that may still play a crucial role in the functioning or build integrity of the ESP-IDF project.

  The agent leverages a tool called `delete_files` to perform actual file removals. The tool expects a set of string of relative file paths and handles the deletion process safely, returning the names of the successfully deleted files.

- **AddEditFilesAgent**: its role is to add or modify files within an ESP-IDF project. It operates based on the structural directives provided by the *FileOrganizerAgent*, and ensures that all newly created or altered files are syntactically valid, complete, and ready for inclusion in the build system.

  Although the agent generates the full contents of each file, it does not directly save files, instead, it forwards them to the *CodeReviewerAgent* for validation and final write operations.

- **CodeReviewerAgent**: it is accountable validating, correcting, and saving code files received from the *AddEditFilesAgent* or *CodeFixerAgent*. It serves as the final gatekeeper before files are persisted to disk, ensuring correctness and maintainability. During its work, the agent checks that the code is both syntactically and logically correct, following the ESP-IDF development conventions and good practices.

  In case it detects errors or omissions, the agent intervenes directly by applying the necessary changes before saving. If there are big errors in some generated files, it can ask to the sender agent to regenerate them. Each corrected or validated file is then saved to the file system using the `generate_code` tool, that actually writes files to the output directory.

- **BuildAgent** it is charged with the compilation phase of the ESP-IDF project, whose main goal is to make an effective build process to verify the correctness of the modified project.

  This agent uses the `build_project` tool that runs the `idf.py build` command within the correctly initialized ESP-IDF environment. When called, the agent builds the project in an isolated environment. At the end of the process, if the build was successful, the agent sends a confirmation message that allows a direct transition to the *ApproveAgent* that closes the conversation. Otherwise, it returns the error log, `stderr`, to assist the *CodeFixerAgent* in diagnosing and resolving the part that triggered the error.

- **CodeFixerAgent**: this agent is responsible for modifying the code when the build attempt fails. Given the project modification by the *AddEditAgent* and especially by the feedback produced by the *BuildAgent*, it performs targeted corrections to resolve the errors. Corrections are made in compliance with ESP-IDF v4.4.8, following the conventions of its folder and file structure. Additionally, the agent ensures proper code formatting, commenting, and the presence of required definitions.

The output is returned as a structured JSON list of modified source and header files. These modifications are not directly saved to the final project state, instead, they are sent in the chat and then the *CodeReviewerAgent* is in charge of validating the changes before saving them.

- **ApproveAgent**: the aim of this agent is only to close the conversation. Its activation happens only when a successful build occurs. The responsibility of this agent is to acknowledge the successful conclusion of the development pipeline and terminate the session using the `approve_saved_files` tool. By explicitly closing the conversation, the agent guarantees that no further actions will be performed.

Coordination and selection of agents

In this system, the agent selection logic is handled by *SelectorGroupChat*, which ensures that only one agent can speak at a time, depending on context and the current task. As previously, a selector function, `selector_func`, has been implemented on the team. This function is used to decide who should speak, in that case, the *PlanningAgent* and one of the other agents alternatively speak, this ensures to follow the workflow strictly.

In Figure 3.4 the workflow for editing generated projects is illustrated. The diagram uses green arrows to represent the standard progression between agents, while orange arrows represent alternative paths triggered by build failures. Red arrows indicate the presence of critical issues in the files that demand to be regenerated. The purple boxes denote specialized tools invoked during the workflow, and the blue arrow shows the interaction with external data sources required by those tools.

The workflow begins with the *PlanningAgent*, which orchestrates the overall process by breaking down high-level objectives into discrete, manageable tasks. These are then delegated to specialized agents in a predefined sequence. The first operational step involves the *UploadProjectAgent*, which uploads all project files from the output directory to make them accessible within the chat environment, using its tool. Once the project is available, the *UnderstandProjectAgent* is responsible for analyzing the uploaded codebase, identifying key components and detecting missing or inconsistent elements. This analysis provides the necessary context for the subsequent agent in the chain. Following this, the *FileOrganizerAgent* determines the appropriate file organization, evaluating which files are essential, which can be safely removed, and which new files need to be added to fulfill the required functionality. The agent outputs three distinct sets of file actions: files to keep, delete, and add. Based on these instructions, the *DeleteFilesAgent* is triggered to remove redundant or obsolete files from the project. The *AddEditFilesAgent* generates the content of the new files and modifies the existing files indicated by the *FileOrganizerAgent*, ensuring they are consistent with the ESP-IDF v4.4.8 conventions. These newly generated and edited files are then passed to the *CodeReviewerAgent*, which performs a validation step: it checks the correctness of the code, verifies compatibility with the ESP-IDF version, ensures adherence to best practices, and applies corrections if needed before saving the files to the project directory; if there are unrecoverable errors, the agent asks to regenerate them to the *AddEditFilesAgent*. Once all files are saved correctly, the *BuildAgent* initiates the compilation of the project. If the build completes successfully, the workflow advances to its final phase. However, if the build fails, the process enters a recovery loop: the *CodeFixerAgent* is activated to inspect build errors and propose or apply necessary modifications to the source code. These fixed files are again reviewed and saved by the *CodeReviewerAgent*. This iterative loop between building and fixing continues until the project is built successfully. At last point, the *ApproveAgent* is called to confirm that all

steps have been completed correctly and to close the conversation.

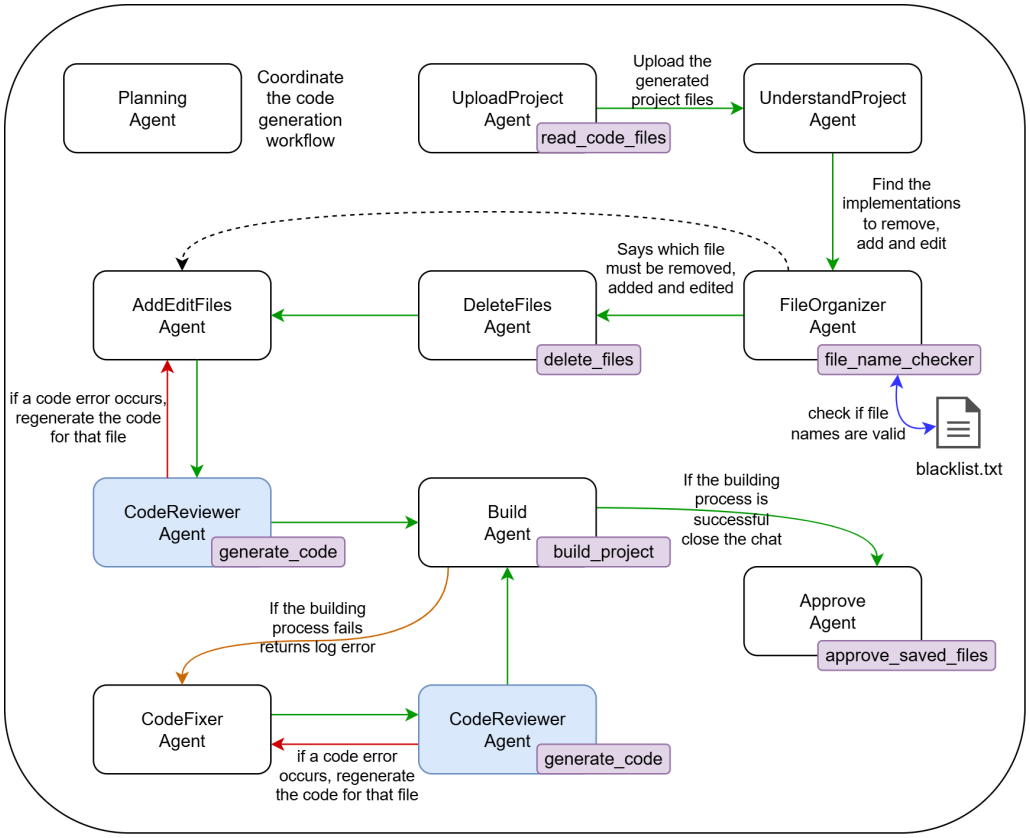

Figure 3.4: Multi-agent chat modification workflow for generated project

Each agent operates autonomously within its domain of expertise, yet the sequential and structured delegation enforced by the *PlanningAgent* ensures orderly progression and avoids interference among tasks. This turn-based interaction enhances transparency, enabling users to trace each decision and transformation applied to the project.

The result is a reliable and traceable editing workflow capable of adapting ESP-IDF projects while preserving structural integrity. The built-in validation mechanisms ensure that only functional and fully compilable code reaches the deployment phase, minimizing the risk of inconsistencies or broken builds on the ESP32 board.

3.4.4 Multi-agent chat for editing task of an existing project

In this section, the system that allows editing for existing ESP-IDF projects is described. Its task is to modify a project from a given user task, so features can be added, modified, or removed from the project. This architecture is a remodeling of the previous one, because few agents have been added and modified, and it is aimed only at large projects. The distinction of these projects has been discussed in Section 3.1.3.

General architecture

The system architecture to modify existing projects, as the other, is composed of multiple agents, and the interaction between them is orchestrated through the *SelectorGroupChat* construct. That regulates the speaking turn and dynamically selects which agent should act based on the state of the conversation and the progress of the task.

In this case, the user loads its ESP-IDF project that they want to edit, and it is assumed that this project works correctly. Unlike the previous system, for editing generated projects, the entire ESP-IDF project is not immediately loaded into the conversation. Instead, only relevant files are selectively retrieved and injected into the chat context. This selective loading approach enables the system to operate within token constraints while still maintaining a persistent and coherent conversation.

As before, due to the limited and well-scoped nature of the editing tasks, the chat session does not require resets or structural rebuilds, as might be necessary in more disruptive workflows such as build-and-fix. However, a recovery mechanism is still integrated; if a build error is introduced as a result of user-requested changes, the system activates a feedback loop that analyzes the error and attempts to apply corrective changes, using the same principles as the build correction phase in the generation pipeline. Nevertheless, since the starting point is an already working project and the modifications tend to be incremental, the likelihood of encountering severe errors or needing deep structural corrections is considerably lower.

Finally, at the end of this section, we will examine a variant of this architecture, which introduces a new agent and modifies the behavior of an existing one to better accommodate complex editing scenarios.

Agents

- **PlanningAgent**: it is the central agent responsible for coordinating the entire editing process of an existing ESP-IDF project. Its main function is not operational, but strategic, it takes care of breaking down a complex goal requested by the user into an ordered sequence of subtasks, delegating them to the various specialized agents.

  This agent does not generate or modify code directly, nor is it authorized to save files. The *PlanningAgent* interacts with a heterogeneous team of agents, each of which is responsible for a specific task in the change lifecycle. Thanks to its coordination, ensures the consistency of the entire change process, minimizing the risk of errors, and optimizing collaboration between specialized agents.

- **UploadHeaderFilesAgent**: Its task is to upload the header files (.h) present in the project to the chat, thus providing an initial overview of the code structure and the APIs available to other agents. It has a predominantly informative role and does not intervene in the direct modification of the code.

  Thanks to the `read_header_files` tool, the agent analyzes the content of the project directory, automatically excluding irrelevant files and folders, such as build directories, and isolates the header files. These are then read with the aim of extracting the function declarations in order to obtain a summary of the type of interfaces exposed by the project's modules.

  In addition to the synthetic information on the header files, the agent also provides a complete list of all the files present in the project. This helps subsequent agents to orient themselves in the code structure and understand which files it will be appropriate to intervene on to satisfy the user's request.

  The end result is a structured message that summarizes both the declared APIs and the files present, providing a solid foundation to start the subsequent agent-driven analysis and transformation phase.

- **UploadProjectFilesAgent**: Its function is to determine, based on the interfaces declared in the .h files and the user's request, which project files are needed to make the requested change.

  The agent operates selectively, avoiding loading the entire project into the chat. This approach was chosen to keep the conversational context contained within the token limits, since in large projects the complete loading would be inefficient or even impossible.

  To extract the relevant files, the agent uses the `read_project_files` tool, which allows reading the content of the files specified in input. In the input it writes the list of the needed files, which allows loading multiple files at the same time in a structured and efficient way. Each file is then returned with its header indicating its path, followed by the full content.

  The agent not only loads the files, but selects them in a reasoned way, analyzing the declarations already present in the header files, comparing them with the user's request, and identifying the portions of the code that need modification or expansion. This includes both files where new features need to be implemented, and files where new features need to be integrated through function calls or logical updates.

  The *UploadProjectFilesAgent* acts as an intelligent filter between the entire code base and the conversational context, ensuring that only the necessary code is uploaded and analyzed, thus laying the foundation for a targeted and efficient change of the project.

- **FileOrganizerAgent**: this agent agent is responsible for defining the file structure of new or modified files of the ESP-IDF project during the modification phase, acting as a central point for the logical organization of the code. Its task is to determine which new files should be created, which should be kept, and, if necessary, which can be removed, based on the user's request and the structure of the project.

  It establishes the location and purpose of each file, so that subsequent agents can operate in a consistent and organized way. To ensure that new file names do not conflict with the ESP-IDF libraries, the agent uses the `file_name_checker` tool. This tool takes as input a list in JSON format of the objects that represent the files to add, each with its path, then analyzes the proposed names and compares them with a predefined blacklist. If it detects non-admitted names, it returns a message that identifies the problematic files, requesting their renaming. Only after having successfully passed this check, the agent can proceed.

  Finally, the agent can also suggest the deletion of obsolete or unused files, but does not directly perform the deletion: it entrusts this decision to the dedicated agent, after verifying the actual uselessness of the file.

- **DeleteFilesAgent**: it is in charge removing files that are no longer needed within an ESP-IDF project, following the instructions provided by the *FileOrganizerAgent*. Its role is part of the project restructuring phase, where, in order to maintain a clean and coherent organization, it is sometimes necessary to eliminate redundant, obsolete or unused components.

  DeleteFilesAgent has decision-making capabilities, it does not automatically execute all deletion requests, but evaluates the actual need to remove the reported files. If it believes that a file is still required for the correct functioning of the project, the agent refrains from deleting.

To perform deletions, the agent invokes the `delete_files` tool, which accepts as input a string containing the relative paths of the files to be removed. The tool checks the physical existence of each file in the filesystem and, if present, proceeds to delete it. At the end of the operation, it returns a message that confirms the list of files successfully deleted.

This agent ensures that project cleanup is done in a safe and controlled way, avoiding accidental deletion of files that are still in use. It also always produces consistent and verifiable output, useful for tracking changes in the multi-agent workflow.

- **AddEditFilesAgent**:it creates and edits files into an ESP-IDF project, precisely following the structural instructions provided by the *FileOrganizerAgent*. This agent is solely responsible for generating and preparing the code, while the responsibility for saving the files and for syntax validation is delegated to the *CodeReviewerAgent*.

  Once it receives the instructions from the *FileOrganizerAgent*, it generates the entire content of each file, including headers and .c implementation, and submits them for saving. It is required to generate the entire content of the files.

  It does not have to modify the `CMakeLists.txt` file, because automatic inclusion of new files is expected by the system. The need to add this implementation is due to the fact that in existing projects the `CMakeLists.txt` file is very long and this could cause an excess of tokens if the whole file was written.

  To ensure project consistency and prevent conflicts, the agent produces its output in JSON format, structured as a list of objects, each of which specifies the name of the file to be created or modified and the complete code it contains.

- **CodeReviewerAgent**: this agent is responsible for reviewing and validating the source code intended to be integrated into the ESP-IDF project. It comes into action whenever another agent, such as the *AddEditFilesAgent* or the *CodeFixerAgent*, generates or editing a file, ensuring that the content produced complies with the technical and stylistic standards imposed by the ESP-IDF framework.

  The primary function of this agent is to deeply analyze the code received, verifying its syntactic correctness. If the code presents anomalies or does not comply with the expected guidelines, the agent is able to intervene directly by correcting the errors and modifying the files automatically, before they are saved. If, on the other hand, the code is already correct, it is immediately validated and saved without alterations. The actual saving of the files occurs through the use of the `generate_code` tool, which not only writes the content to disk in the correct position, but also takes care of automatically updating the CMakeLists.txt files.

- **BuildAgent**: the agent performs the compilation procedure using the official ESP-IDF environment, using the `idf.py build` command inside the modified project directory. If errors occur during the compilation, the agent returns a failure message, the error logs. This log is then used to feed back the system: the error is passed to the *CodeFixerAgent*, which will take care of identifying and fixing the problems found.

  If the compilation ends successfully, the *BuildAgent* confirms that the project has been built correctly and signals to the *ApproveAgent* that the process can be concluded. In this way, the agent acts as a final checkpoint to validate that all the changes made are formally correct and that the code can be integrated without causing compilation errors.

This automatic validation phase is essential to ensure the stability of the project and allows the multi-agent system to operate reliably, minimizing the need for human intervention in the firmware development and maintenance cycle.

- **CodeFixerAgent**: it only acts in case of an error during the compilation phase. This agent is responsible for analyzing the error message provided by the *BuildAgent* and making the necessary corrections to the code to solve the problem. Once the error log is received, the agent identifies the causes, such as: missing functions, undefined constants, syntax errors or violations of the project structure. After correcting the code, it returns it in the form of JSON objects.

  It is important to underline that the *CodeFixerAgent* does not directly save the files in the project, once the changes have been made, it sends the new code, which *CodeReviewerAgent* provides to do the final validation and saving. This intermediate step provides an additional level of control, improving system reliability and preventing the introduction of recurring errors.

- **ApproveAgent**: Its role is simple but essential, approving and closing the process once the project has been successfully built. The agent is only activated if the *BuildAgent* has confirmed that the project has been built correctly, meaning that there are no errors and that all changes introduced are compatible with the ESP-IDF infrastructure.

  When a successful build happens, the *ApproveAgent* uses the `approve_saved_files` tool to approve and close the review and change cycle. This event marks the completion of the automated process, with the project in a stable state and ready for use or release.

Coordination and selection of agents

As mentioned above, the agent selection logic is handled by *SelectorGroupChat*, ensuring that only one agent can speak at a time, depending on the context and the current task. The implemented `selector_func` allows to alternately speak the planning agent and one of the other agents, letting the workflow to be followed more strictly.

Figure 3.5 represents the workflow to modify existing projects. In the schema, the green arrows show the standard path between agents, the orange arrow illustrates an alternative path to go in case of build failures, and the red arrows specify the presence of critical issues in the files that request to be regenerated. The purple boxes are specialized tools that can be invoked by agents during the workflow, and the blue arrow represents the connection between external data and tools.

The workflow starts with the *PlanningAgent*, which acts as the central coordinator of the entire process of modifying and adapting an existing ESP-IDF project. This agent is responsible for breaking down the complex goals requested by the user into an ordered and manageable series of subtasks, delegating each task to a specific specialized agent. Each operation is punctuated by a well-defined sequence, which ensures the correct progress of the flow and prevents the activities from overlapping or being executed prematurely.

The first concrete step involves the *UploadHeaderFilesAgent*, which is in charge of uploading the project header files, specifying only the functions declared inside and the list of all project files in the conversation. Subsequently, the *UploadProjectFilesAgent* analyzes the files just uploaded together with the user request to determine which other files in the project are relevant to the editing operation. This preliminary selection allows the context to be reduced to the minimum necessary.

Once the initial collection and analysis phase is completed, the *FileOrganizerAgent* intervenes, which evaluates the structure of the project and determines its optimal organization. In particular, it identifies the files to keep, those to delete, and those to create from scratch to satisfy the new features. Starting from these indications, the *DeleteFilesAgent* removes unnecessary files from the project, while the *AddEditFilesAgent* generates the content of the new files and the files to be modified, necessary for the requested extension. The newly created and edited files are not saved immediately, they first pass through a quality control entrusted to the *CodeReviewerAgent*, which verifies the correctness of the code and the coherence with the general architecture of the project. The files are saved only after this validation.

Once all changes have been integrated and validated, the *BuildAgent* compiles the project using `idf.py build`. If the build is successful, the flow continues to its conclusion. Otherwise, the *CodeFixerAgent* is activated, which analyzes the error messages generated during the compilation and proposes the corrective changes needed to fix the found problems. Again, the corrected code is reviewed and saved by the *CodeReviewerAgent*, after that the project is rebuilt. This cycle of correction and verification can be repeated multiple times until the build is successful.

Once the project is successfully compiled, the ApproveAgent is activated, which formally approves the entire sequence of operations and closes the conversation, marking the end of the flow.

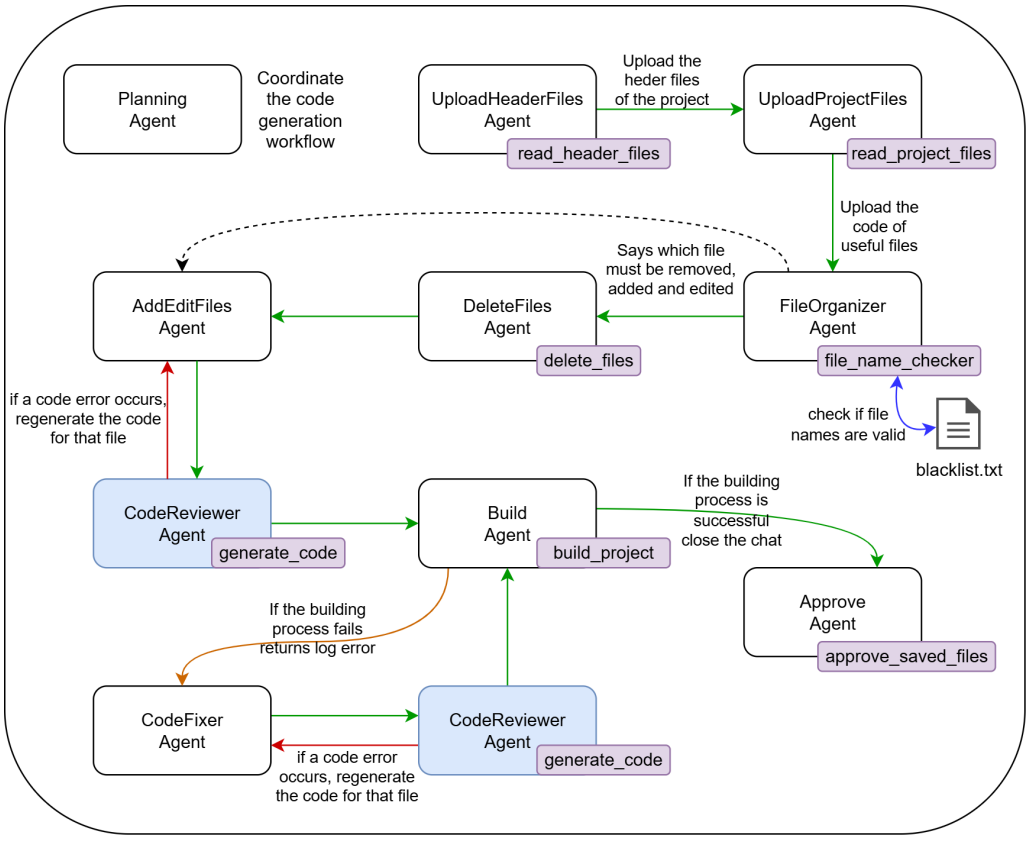

Figure 3.5: Multi-agent chat modification workflow for existing project

The turn-based architecture adopted by this multi-agent system ensures that each agent operates exclusively in its own domain of expertise, in a sequential and traceable way. This approach increases transparency, allowing the user to follow every transformation

applied to the project. The end result is a robust, modular, and reliable workflow, capable of dynamically adapting an ESP-IDF project while preserving its structure and ensuring the compilability of the code before its eventual distribution.

Alternative implementation for existing project modification

An alternative to the main workflow for modifying existing projects has been developed with the aim of improving the efficiency of change requests, especially those involving the addition of new features to an already structured project. This implementation differs mainly in the management of file changes, optimizing the process to reduce token consumption and increase the modularity of the generated code.

In typical change requests of existing projects, the user tends to request the addition of new features rather than a complete revision. For this reason, within the original workflow, even small additions ended up completely regenerating large files. This strategy had two main disadvantages: the increase in the number of tokens processed and the risk of introducing accidental errors by rewriting sections of the already working code.

To address this problem, a clearer separation between two types of interventions has been introduced, the creation of new files containing the logic of the new features, and the targeted modification of existing files, necessary only to integrate the new functions through simple calls. This approach is better suited to the incremental nature of changes, allowing for more localized and precise work. In particular, the agent originally responsible for generating and modifying files, the *AddEditFilesAgent*, has been refactored to focus almost exclusively on creating new source files. In parallel, the *MergeCodeAgent* has been introduced, an agent responsible for the targeted insertion of new function calls into existing files. This agent receives as input the name of the function to be integrated, the specific context in which to insert it, the target file, and the includes to be added, generating only the lines needed to complete the integration.
This division brings some advantages:

- It significantly reduces the number of tokens used to describe and process changes to files, since it is not necessary to resend the entire content of the file to be modified.
- It minimizes the risk of unwanted alterations in existing code, limiting changes only to the required insertion points.
- It improves the scalability of the system, allowing complex projects with large files to be handled more efficiently.

As mentioned above, to support the new approach described in the previous section, structural changes are needed in two fundamental agents of the system.

- The **AddFilesAgent** agent has been refocused mainly on creating new files. The generated files are then validated and saved by the *CodeReviewerAgent*, as in the traditional workflow.
- The **MergeCodeAgent** was introduced to fill the gap left by the specialization of *AddFilesAgent*. Its task is to integrate new features into existing files, in a minimal and targeted way. The agent acts only after the generated files have been approved and saved by the *CodeReviewerAgent*, avoiding the introduction of unsaved code. This agent does not rewrite the entire content of the files, but limits itself to insert calls to generated functions into the new files, add the necessary `#include` directives

only if not already present, insert the source code in the correct place within the existing file, identified by a block of 10-20 lines called insertion context.

The agent behavior is supported by the `insert_code` tool, which receives as input a JSON array containing all the insertion instructions. Each JSON object must specify `file_name`, relative path of the file to modify; `context`: the lines of code immediately preceding the insertion point; `insertion_code`, code to insert immediately after the context; `includes`, any header files to include. From this given input, the function can add the code in a specific position inside the file including headers file, this allows for a clean and safe integration, keeping the original file intact except for the strictly necessary lines.

In Figure 3.6, the workflow with *MergeCodeAgent* is shown. The *MergeCodeAgent* enters the workflow exactly after the files generated by *AddFilesAgent* have been validated and saved successfully by the *CodeReviewerAgent*. The flow remains almost unchanged compared to the main workflow, Section 3.4.4, with the only addition of this intermediate step. The updated sequence is the following, *AddFilesAgent* creates the new files containing the new features, *CodeReviewerAgent* verifies and saves these files. *MergeCodeAgent* integrates the new features into existing files. The workflow then continues with the build phase and possible debugging.

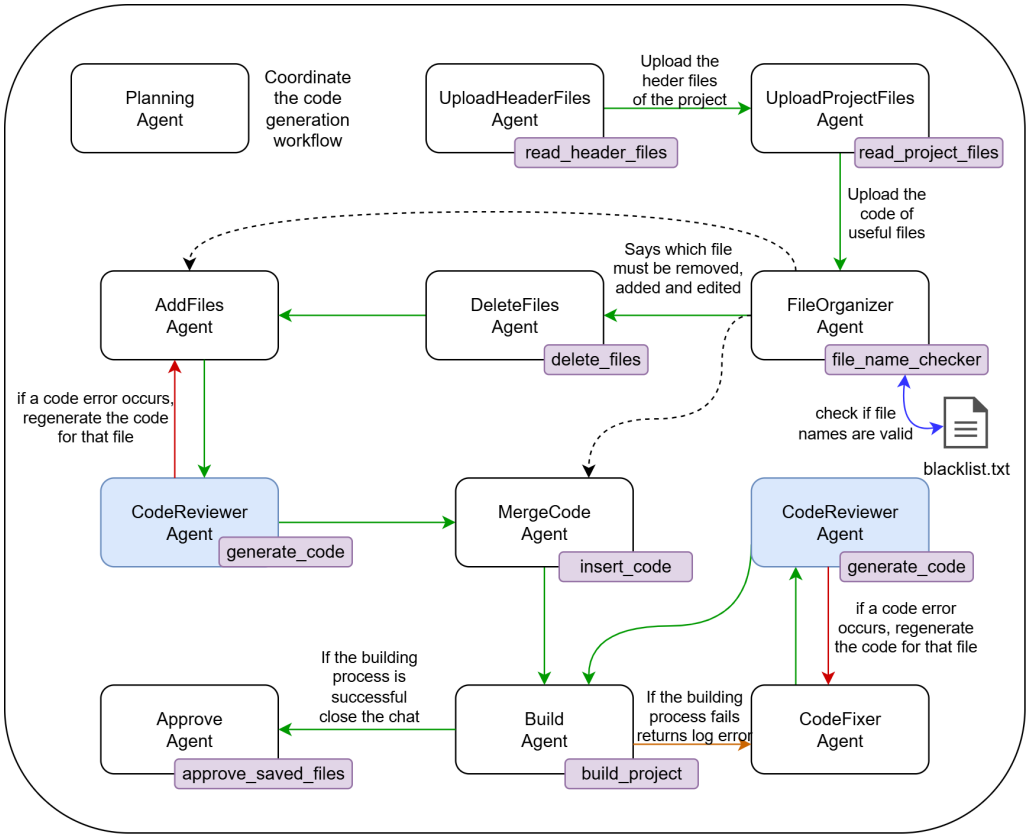

Figure 3.6: Multi-agent chat modification workflow for existing project with MergeCodeAgent

In summary, this second implementation is particularly suitable for use cases where existing projects need to be extended with new features while maintaining the core of the original project. Thanks to the functional distinction between agents dedicated to creation

and those dedicated to modification, it is possible to obtain a more robust, modular, and suitable workflow for incremental changes.

Chapter 4

Results and discussion

In this chapter, the results obtained from the different approaches implemented for the generation and modification of ESP-IDF projects will be discussed. The generation and modification times of the predefined tasks will be analyzed, and the effectiveness of each architecture in terms of performance and reliability will be evaluated. The analysis will start from the first developed approach, a single-agent chat, up to the more complex systems based on multi-agent architectures.

4.1 Single-Agent Chat

The first approach implemented consists of a chat with a single general-purpose agent, capable of generating ESP-IDF code in response to a user request. This solution stands out for its extreme speed, with almost instantaneous responses, and for its great flexibility in handling generic or fragmented requests.

Unfortunately, it also has some limitations. The code produced is not structured as a complete project, but is limited to isolated fragments, often lacking the context necessary for direct compilation. Furthermore, there is no guarantee that the code will be functional; in fact, there is no mechanism for validation, testing, or persistent saving. This makes the solution unsuitable for complex or production-oriented tasks.

Being an initial phase of the work, this approach has not been subjected to an effective compilation or deployment tests. It could be useful in exploratory or educational scenarios, where the user wants to obtain an example of rapid implementation. It must be highlighted that, thanks to integration with RAG systems, the agent is able to provide semantically coherent responses even in the absence of a structured project.

4.2 Initial multi-agent chat for project generation

This system has been the first step of the real multi-agent architecture implemented. As discussed in Section 3.4.1, its aim is to generate the project from a given user task, leveraging a sequence of specialized agents.
But in this approach, no validation or build process is performed after the project generation. All project files are simply saved and returned to the user, who is then responsible for testing and verifying its correctness. This means that although the architecture provides complete and structured ESP-IDF projects, the reliability of the result is not guaranteed.
To evaluate the capabilities of the system, a variety of representative tasks were used. These include:

- *Generate an I2C driver for ESP32.*
- *Generate an ESP-IDF project for ESP32 that reads the status of a CNC machine via digital inputs (e.g., start, stop, alarm) and sends the data in real time to an MQTT server. Simultaneously, the system must expose the same data via BLE as a GATT server accessible by a mobile app. Handle WiFi and BLE coexistence, prioritizing MQTT connection stability.*
- *Generate a project that allows connecting to IoT HUB*
- Further tasks of similar complexity have also been considered.

Despite the inherent stochasticity of the model, in many cases the generated projects were functional and complied with the user request. But sometimes some common issues were observed when compiling:

- Missing `#include` directives for required libraries.
- Incorrect library names in include statements.
- Undeclared configuration variables (typically of the form `CONFIG_...`) used in the code.

The integration with the RAG system has made the generated code logically and contextually appropriate, but still prone to small implementation oversights. The time saved compared to writing a project manually is a significant advantage of this approach. The automatic generation of a structured ESP-IDF project in just a few minutes represents a valuable support for developers, especially in the prototyping phase. But it is necessary to point out that manual adjustments are often required due to the typical issues described above. These post-generation fixes, such as correcting include statements or adding missing configuration variables, can introduce additional overhead and must be taken into account when evaluating the overall effectiveness of this system.

4.3 Multi-agent system for project generation tasks

As described in Section 3.4.2, this architecture is a more robust and complete implementation which makes sure that the final output is a functionally correct ESP-IDF project, fully aligned with the task described by the user. Introducing a significant improvement; after project creation, it is automatically validated through an actual compilation step. If any build errors are detected, the system does not return the project immediately to the user. Instead, the errors are passed back to the agent that fixes the code, which iteratively corrects the problematic files until the compilation is successful. This feedback loop continues automatically until the project is confirmed to be valid and buildable. The result is that the user receives a project that is guaranteed to be compilable, implementing the required features as defined by the original task prompt.

To verify the actual correctness and functionality of the final projects,not just syntactic or compilation-level correctness, each project has been flashed onto a physical ESP32 board and tested. This has allowed for concrete validation of the software behavior and interaction with hardware peripherals, like GPIOs, I2C, BLE, MQTT.

Table 4.1: User tasks and average project generation time using the final MAS

User task	**Avg. time to generate**
Generate an I2C driver for the ESP32.	8 min 36 s
Generate an ESP-IDF project for the ESP32 that implements a remote monitoring system for temperature and humidity using a DHT22 sensor. The data must be sent to an MQTT server every 10 seconds. The code must include Wi-Fi connection management, error handling, and automatic reconnection in case of disconnection.	11 min 20 s
Generate an ESP-IDF project for the ESP32 that implements a weather station with multiple sensors (DHT22 for temperature and humidity, BMP180 for atmospheric pressure). The data must be collected and sent to both an MQTT server and an HTTP REST server every 10 seconds. The code must include: • Wi-Fi connection management with automatic reconnection. • Error handling for sensor reading or server connection failures. • Configuration via NVS to allow setting network parameters without recompiling the code.	13 min 42 s
Generate an ESP-IDF project for the ESP32 that reads the status of a CNC machine via digital inputs (e.g., start, stop, alarm) and sends the data to an MQTT server in real time. Simultaneously, the system must expose the same data via BLE as a GATT server readable by a mobile app. It must handle concurrent use of Wi-Fi and BLE, prioritizing MQTT connection stability.	13 min 18 s
Generate an ESP-IDF project for the ESP32 that acts as a gateway between Modbus TCP (via Ethernet or Wi-Fi) and Modbus RTU (via UART). The device must be able to configure register addresses and mappings from a JSON file located on an SD card. Include Wi-Fi reconnection and RTU bus management (timeouts, collisions).	12 min 36 s

Continued on next page

User task	Avg. time to generate
Generate an ESP-IDF project for the ESP32 that reads industrial gas sensors (e.g., CO, CH4) via ADC and sends the data every minute to a remote HTTPS server. If the Wi-Fi connection fails, the system must provide the data via BLE GATT to mobile devices until the connection is restored. Sign the data using SHA256.	14 min 55 s
Generate an ESP-IDF project for the ESP32 that implements a simple LED lamp control system via MQTT commands. The device must connect to a Wi-Fi network configured in the code and subscribe to an MQTT topic (e.g., 'home/lamp/control'). If it receives the message "ON", it should turn on the LED connected to GPIO 2; if it receives "OFF", it should turn it off. The project must include: • Wi-Fi connection with automatic reconnection. • MQTT broker connection with disconnection handling. • UART logging of the lamp status.	15 min 20 s
Create a project that connects to an IoT hub.	9 min 27 s

As shown in Table 4.1, the average project generation time is around 12 minutes, depending on the complexity of the task. This represents a potential time saving compared to manual development, especially for complex projects that would otherwise require significant planning and coding effort.

This system is particularly effective for tasks that involve multiple components or peripherals, where the architectural setup, configuration, and correct use of ESP-IDF libraries can be complex. In contrast, for smaller tasks requiring the creation of only one or two files, the generation time may feel less efficient given the fixed overhead introduced by the compilation and validation cycles.

It is important to note that compilation times, each taking approximately 3 minutes, can heavily impact the overall generation duration, especially when multiple iterations are required to reach a successful build. Although, this trade-off offers the guarantee of a fully functional ESP32 project at the end of the process.

4.4 Multi-agent system for editing generated project

This experimental phase concerns the modification of generated projects through the multi-agent system, as described in Section 3.4.3. This approach allows to apply new features to an already working generated project, ensuring that the resulting project is correctly compilable and compatible with the ESP-IDF structure.

Similarly to the complete generation, also in this case a validation mechanism through coverage has been maintained, but unlike the generation system, it is done in the same chat without the need to recreate a new chat to build and fix it. So, it can return to the

user a project that can be immediately tested. After compilation, the modified projects have been flashed on the ESP32 board to test their correct execution in a real environment, observing the interaction between the new features and the software components.

Table 4.2: User tasks and average project modification time using the MAS for editing generated projects

User task	**Avg. time to edit**
Change the MQTT publish interval from 10 to 30 seconds.	3 min 32 s
Replace the DHT22 sensor with a BME280, maintaining similar functionality.	4 min 11 s
Add an LED that blinks every time an MQTT message is sent.	3 min 38 s
Add UART printing of the WiFi signal strength (RSSI) every 30 seconds.	3 min 55 s
Add the ability to configure SSID and WiFi password via UART on first boot.	3 min 49 s
Include the option to change the MQTT topic via UART.	3 min 54 s
Implement TLS encryption in the MQTT connection.	3 min 58 s
Add saving of the last data read from the DHT22 to NVS.	4 min 05 s
Add a watchdog timer to reboot the device in case of a freeze.	3 min 46 s
Add a function that reads 2 bytes from the I2C device at address 0x40 and prints the values via UART every second.	3 min 34 s

In Table 4.2 some examples of requested modifications and the related average completion times are reported, the project used for the edit tasks reported is the project generated from the task *Generate an ESP-IDF project for the ESP32 that implements a remote monitoring system for temperature and humidity using a DHT22 sensor. The data must be sent to an MQTT server every 10 seconds. The code must include Wi-Fi connection management, error handling, and automatic reconnection in case of disconnection.*.

As can be seen, the average completion times are around 4 minutes, significantly lower than the complete generation of the project. This is mainly due to the fact that since these are incremental changes, the probability that the compilation will be successful on the first attempt is much higher, thus reducing the need for repeated correction and rebuild cycles.

Since a single compilation takes around 3 minutes, it can be said that, in many cases, the entire process is completed in a single cycle, making this modality particularly efficient for functional extensions to existing projects.

This multi-agent system has shown to be not only valid in the complete generation of ESP-IDF projects, but also particularly effective for their modification. It is able to integrate or modify features in generated projects in a fast, safe, and verifiable way, significantly reducing the cognitive and operational load for the developer.

4.5 Multi-agent system for editing existing project

In this last evaluation phase, the ability of the multi-agent system to perform modifications on existing large ESP-IDF projects was analyzed. Unlike the previous cases, here the goal is not to generate a project from scratch or modify a recent one, but to work on a pre-existing, structured code rich in already implemented features.

It is important to note that the project used for this experimentation was already equipped with almost all the features expected from the ESP-IDF platform, including Wi-Fi management, MQTT, BLE, NVM, hacl, UART, error handling, automatic reconnection, and many others. This makes it an ideal testbed to evaluate the effectiveness of incremental changes, since it represents a realistic and complex context.

As highlighted in Section 3.4.4, the initial architecture showed some limitations mainly due to the overall size of the context. In fact, since during the modification phase it was necessary to provide the agent with the entire content of the files involved, the maximum limit of tokens that the model could handle was often reached. This represents a significant obstacle in large-scale projects, which can contain dozens of files, hundreds of functions, and extensive dependencies.

Two tasks were used to test this scenario:

- *Manage the flash saving of logs in the data_stg partition, max size 2MB, with automatic log rotate management.*
- *Encrypt and decrypt hacl messages, using the enc_test and dec_test functions in the code respectively.*

In both cases, the changes were successfully applied and tested on the ESP32. However, due to the initial system structure, a very high average change time was found, in some cases exceeding 30 minutes. This is mainly due to the fact that each compilation cycle takes between 15 and 20 minutes, due to the size of the project and the complexity of the code.

4.5.1 Architecture with MergeCodeAgent

To address the above limitations, a new architecture, described in Section 3.4.4, has been designed and adopted, introducing the `MergeCodeAgent` component. This agent has the specific task of inserting new features into existing files, avoiding having to rewrite the entire file for each change. The mechanism is based on the extraction of a limited context for the targeted insertion of the new code, significantly reducing the probability of exceeding the token limit. This architecture has proven to be particularly effective for extensive tasks, where the user asks to add new features.

Even in this configuration, the average modification times remain around half an hour, since the build process still requires 15 to 20 minutes, and in some cases multiple compilation cycles are necessary. However, the new architecture substantially improves the scalability of the system, allowing the effective modification of complex and articulated projects.

Chapter 5

Conclusion

The main goal of this thesis was to explore and develop a system capable of automatically generating code for edge IoT devices, specifically the ESP32 platform, starting from simple natural language descriptions. The underlying idea is to leverage the potential of artificial intelligence, particularly large language models (LLMs), to actively support software development in embedded contexts.

The work evolved incrementally, starting from a single agent capable of generating code fragments in response to generic requests, to a more complex multi-agent systems capable of producing entire projects, verifying their correctness through compilation cycles, and modifying them consistently even after the initial generation.
In particular, the following systems have been designed and tested:

- **Single-agent**: it is characterized by extreme speed of response and flexibility. It leverages RAG techniques to incorporate contextual information into code generation, but is limited to the generation of code fragments that lack verification and design structure.
- **Multi-agent for project generation**: this system is capable of creating complete ESP-IDF structures in a few minutes, but without an automatic post-generation validation process. The code quality is generally good thanks to the integration with RAG systems, but minor errors may occur that require manual interventions, like adding missing configurations or including required libraries in the project files.
- **Multi-agent with automatic validation**: it is a complete system that automatically generates, verifies, and corrects the project until a successful compilation is achieved. Average generation times are around 10 minutes, mainly influenced by build cycles (3 minutes each), but they guarantee a working project ready for deployment on ESP32.
- **Multi-agent for editing generated projects**: allows the functional extension of already created projects, with average times of about 4 minutes. This approach has proven to be very efficient since, being incremental changes, the code often compiles correctly on the first try.
- **Multi-agent for editing existing projects**: developed to deal with the complexity of large projects, with many dependencies and files. The initial architecture showed limitations due to exceeding the maximum number of tokens that the system could handle, leading to editing times exceeding 30 minutes, mainly due to the duration of the compilation processes (15-20 minutes each).

- **Optimized architecture with `MergeCodeAgent`**: it is an extension of the previous system. It introduced to reduce the risk of exceeding the token limit and improve scalability. This new architecture has proven to be particularly effective for complex tasks that require the addition of new features, while maintaining editing times similar to the previous configuration.

In general, the developed system has demonstrated that it can effectively automate different aspects of software development for embedded devices, both in the generation and modification phases. The main benefits observed include:

- Significant reduction in development time for complex tasks.
- Possibility to obtain working and verified projects starting from simple textual descriptions.
- Efficient support for modification and refactoring of existing code, even in highly complex scenarios.

Naturally, there is still room for improvement. Compilation times, especially for large projects, represent a significant bottleneck, especially when multiple verification cycles are involved. Furthermore, context and token management must be carefully balanced to ensure stability and performance in complex tasks.

Another area of improvement concerns the selection of relevant files when modifying existing projects; currently, the system adopts an automatic strategy that can be imprecise in very complex contexts. It would be desirable to improve the mechanism for identifying the files to be provided to the agent, for example, with more refined semantic selection techniques or by involving the user in the explicit specification of the files involved. This would reduce the risk of exceeding token limits and ensure a more targeted and effective context.
Some future developments could include:

- The introduction of incremental or parallel compilation mechanisms to reduce verification and build times.
- Integration with automatic testing systems and assisted debugging.
- Extension of the architecture to different domains, such as automatic generation of websites, using specialized agents for the production of HTML, CSS, JavaScript files or React components, also through the use of a RAG system built on documentation and specific knowledge of the language and framework used.
- Integration of technical documentation as input, like specifications provided by companies, to allow the generation of projects directly from requirements documents or technical manuals.

It can be said that the thesis demonstrates how artificial intelligence, combined with a well-designed multi-agent architecture, can represent a valid support for embedded software development, opening up new possibilities for the assisted design of IoT systems in the edge environment.

Bibliography

[1] Acorn. What is retrieval augmented generation (rag)?, 2024. URL: https://www.acorn.io/resources/learning-center/retrieval-augmented-generation/.

[2] Aisera. Aisera blog multi-agent system. URL: https://aisera.com/blog/multi-agent-ai-system/.

[3] Xavier Amatriain. Prompt design and engineering: Introduction and advanced methods, 2024. URL: https://arxiv.org/abs/2401.14423, arXiv:2401.14423.

[4] CrewAI. Crewai. URL: https://www.crewai.com/.

[5] Digi International. Edge computing vs. cloud computing. URL: https://www.digi.com/blog/post/edge-computing-vs-cloud-computing.

[6] Espressif Systems. ESP-IDF Programming Guide. URL: https://docs.espressif.com/projects/esp-idf/en/latest/.

[7] Espressif Systems. ESP32 Series Datasheet. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[8] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL: https://arxiv.org/abs/2312.10997, arXiv:2312.10997.

[9] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL: https://arxiv.org/abs/2308.00352, arXiv:2308.00352.

[10] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, January 2025. URL: http://dx.doi.org/10.1145/3703155, doi:10.1145/3703155.

[11] IBM. Ai agent orchestration. URL: https://www.ibm.com/it-it/think/topics/ai-agent-orchestration.

[12] LangChain Inc. Langgraph. URL: https://langchain-ai.github.io/langgraph/concepts/why-langgraph/.

[13] IoT Analytics. State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally, 2024. URL: https://iot-analytics.com/number-connected-iot-devices/.

[14] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL: https://arxiv.org/abs/2005.11401, arXiv:2005.11401.

[15] Microsoft. Autogen agentchat examples, 2024. URL: https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/examples/index.html.

[16] Microsoft. Autogen agentchat user guide, 2024. URL: https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/.

[17] Microsoft. Autogen: Enabling next-generation large language model applications, 2024. URL: https://www.microsoft.com/en-us/research/blog/autogen-enabling-next-generation-large-language-model-applications/.

[18] Microsoft Azure. Azure.search.documents - searchclient class. URL: https://learn.microsoft.com/en-us/dotnet/api/azure.search.documents.searchclient?view=azure-dotnet.

[19] Microsoft Azure. Introduction to azure iot. URL: https://learn.microsoft.com/it-it/azure/iot/iot-introduction.

[20] Anuja Nagpal. Ai copilot for the modern developer : Leveraging genai in software development. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 10:702–711, 10 2024. doi:10.32628/CSEIT241051056.

[21] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023. URL: https://arxiv.org/abs/2302.06590, arXiv:2302.06590.

[22] DeepWisdom Team. Metagpt site, 2023. URL: https://www.deepwisdom.ai/.

[23] DeepWisdom Team. Metagpt: The mult-agent framework for engineering tasks, 2023. URL: https://github.com/geekan/MetaGPT.