



Article

Leveraging Retrieval-Augmented Generation for Automated Smart Home Orchestration

Negin Jahanbakhsh ¹, Mario Vega-Barbas ^{1,*}, Iván Pau ¹, Lucas Elvira-Martín ¹, Hiran Moosavi ¹
and Carolina García-Vázquez ²

¹ ETSIS de Telecomunicación, Universidad Politécnica de Madrid, Calle Nikola Tesla S/N, 28038 Madrid, Spain; negin.jahanbakhsh@upm.es (N.J.); ivan.pau@upm.es (I.P.); lucas.elvira@upm.es (L.E.-M.); hirad.moosavi@alumnos.upm.es (H.M.)

² Facultad de Diseño y Tecnología, University of Design, Innovation and Technology, 28016 Madrid, Spain; carolina.garcia@udit.es

* Correspondence: mario.vega@upm.es

Abstract: The rapid growth of smart home technologies, driven by the expansion of the Internet of Things (IoT), has introduced both opportunities and challenges in automating daily routines and orchestrating device interactions. Traditional rule-based automation systems often fall short in adapting to dynamic conditions, integrating heterogeneous devices, and responding to evolving user needs. To address these limitations, this study introduces a novel smart home orchestration framework that combines generative Artificial Intelligence (AI), Retrieval-Augmented Generation (RAG), and the modular OSGi framework. The proposed system allows users to express requirements in natural language, which are then interpreted and transformed into executable service bundles by large language models (LLMs) enhanced with contextual knowledge retrieved from vector databases. These AI-generated service bundles are dynamically deployed via OSGi, enabling real-time service adaptation without system downtime. Manufacturer-provided device capabilities are seamlessly integrated into the orchestration pipeline, ensuring compatibility and extensibility. The framework was validated through multiple use-case scenarios involving dynamic device discovery, on-demand code generation, and adaptive orchestration based on user preferences. Results highlight the system's ability to enhance automation efficiency, personalization, and resilience. This work demonstrates the feasibility and advantages of AI-driven orchestration in realising intelligent, flexible, and scalable smart home environments.

Keywords: smart home orchestration; generative AI; large language models (LLMs); retrieval-augmented generation (RAG); AI agent; OSGi framework; dynamic service bundles; vector databases; IoT integration; AI-driven automation; real-time adaptation



Academic Editor: Paolo Bellavista

Received: 2 April 2025

Revised: 28 April 2025

Accepted: 28 April 2025

Published: 29 April 2025

Citation: Jahanbakhsh, N.; Vega-Barbas, M.; Pau, I.; Elvira-Martín, L.; Moosavi, H.; García-Vázquez, C. Leveraging Retrieval-Augmented Generation for Automated Smart Home Orchestration. *Future Internet* **2025**, *17*, 198. <https://doi.org/10.3390/fi17050198>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The expansion of the Internet of Things (IoT) has accelerated the widespread adoption of smart home devices, profoundly transforming automation and digital interaction in residential, commercial, and industrial settings. As interconnected devices become increasingly prevalent, effectively managing and orchestrating them has become essential for optimising performance, reducing user intervention, and ensuring seamless interoperability. Nonetheless, traditional approaches to smart home automation—often based on predefined rules or limited device integration—have limitations in adaptability and scalability, restricting their capability to predict or respond to dynamic conditions [1]. These limitations have spurred

research into AI-driven orchestration mechanisms that enhance automation flexibility while integrating a wide range of smart devices.

One emerging approach to address these challenges is generative AI, particularly through large language models (LLMs). Owing to their ability to understand and generate text in a human-like manner, LLMs enable more natural interactions, real-time adaptations, and context-rich decision-making. Crucially, their capability to generate code on the fly opens new possibilities for dynamic orchestration: rather than relying on static, predefined workflows, the system can automatically produce or modify service logic in response to user requests and changing conditions. In the context of intelligent environments, this capacity endows systems with the means to interpret user demands, anticipate future needs, and continuously adapt to evolving preferences. Recently, Retrieval-Augmented Generation (RAG) has emerged as a promising technique that links large language models with specialised knowledge repositories, thereby enriching contextual reasoning in automation systems. By combining information retrieval from vector databases or specialised repositories with text generation, RAG allows the system to incorporate relevant, up-to-date data into its decision-making process, delivering more precise and context-aware responses in dynamic IoT settings.

Alongside these AI-driven advancements, the OSGi framework is a robust and widely accepted technology for ensuring modularity and dynamic service management. Thanks to its ability to deploy, update, and manage system components in real time, OSGi provides an ideal architecture for scalable smart home orchestration [2]. Despite the strengths of modern AI techniques and proven frameworks like OSGi, challenges such as heterogeneous device integration, dynamic service updates, real-time adaptation, and cross-device communication persist.

In response to these challenges, this research introduces a novel smart home orchestrator that harnesses the potential of generative AI, specifically LLMs, to dynamically generate service bundles based on user requirements, thereby enhancing both personalisation and efficiency in connected home environments. The proposed system allows users to articulate their needs in natural language, which is then converted into executable orchestration logic. By integrating AI-based retrieval and dynamic service management, the orchestrator increases flexibility and intelligence in smart home environments. The system's performance was validated through multiple use-case scenarios, demonstrating its ability to adapt to changing user needs, optimise device functionalities, and improve overall automation efficiency. As such, this work contributes to the advancement of AI-driven smart home orchestration, laying the groundwork for the emergence of more adaptive and intelligent automation frameworks.

2. Background

Smart home systems, underpinned by the Internet of Things (IoT), have traditionally relied on rule-based or semi-automated solutions, allowing users to configure device behaviours in controlled environments [1]. While these early approaches significantly improved convenience and reduced manual oversight, they often lacked the flexibility to accommodate changing user requirements, device heterogeneity, and evolving environmental conditions. As the proliferation of connected sensors, actuators, and appliances continues to accelerate, the need for more adaptive and context-aware orchestration mechanisms has become increasingly evident [3]. Placing our proposed approach within this evolving landscape calls for examining several foundational concepts. Firstly, understanding how different IoT devices, protocols, and services can be orchestrated into a cohesive environment sheds light on the fundamental challenges of managing interoperability and responsiveness. Building on this, recent advances in generative AI—particularly in code

generation using large language models—highlight new possibilities for automating the creation and revision of system logic in response to changing user needs. However, these models do not operate in isolation, and thus, the technique of Retrieval-Augmented Generation (RAG) provides a means to inject real-time context from domain-specific repositories into the generative process, ensuring that the solutions proposed remain accurate and current. To integrate and manage these dynamically generated or updated services without disrupting the broader system, a robust software architecture is imperative, and the OSGi framework has emerged as a widely accepted technology for ensuring modular and seamless service deployment. Finally, despite the strengths of these approaches, unresolved issues—ranging from security and privacy concerns to device heterogeneity—underscore the need for further exploration. In the subsections that follow, we discuss these pillars of smart home orchestration in detail, thereby laying the groundwork for the design and validation of our proposed system.

2.1. Smart Home Orchestration

One of the primary challenges in realising the full potential of smart home systems lies in orchestrating multiple Internet of Things (IoT) devices, protocols, and services so that they behave collectively as a cohesive and adaptive environment. While simple rule-based methods—such as scheduling lights based on the time of day or triggering alarms under predefined conditions—were sufficient for early smart home deployments, contemporary environments often demand more nuanced coordination [4]. Orchestration, in this sense, involves the continuous and dynamic management of device interactions, event triggers, and resource allocation, aiming to achieve goals such as energy savings, personalised user experiences, and seamless interoperability with minimal direct supervision [5].

Research in this area has progressively moved beyond static or semi-static policies towards more sophisticated paradigms that offer higher degrees of flexibility and autonomy. Agent-based systems, for instance, distribute decision-making across multiple agents capable of learning user preferences and negotiating shared resources [6,7]. Semantic web technologies leverage ontologies to formally represent device capabilities and relationships, enabling reasoning engines to compose services on the fly [8]. Other approaches explore context-aware frameworks in which situational data (e.g., location, weather, or user activity) informs real-time adjustments to device behaviour. Despite these advances, many existing solutions still rely on extensive manual configuration or are unable to accommodate rapid changes without significant downtime or reprogramming.

The next generation of orchestration mechanisms aims to address these gaps through adaptive, AI-driven methods capable of dynamically generating and updating logic in response to changing conditions. Such an evolution holds the promise of not only simplifying system management for end-users but also improving the overall responsiveness and resilience of smart home ecosystems. In the subsequent subsections, we explore how generative AI techniques, combined with retrieval strategies and robust modular frameworks, can pave the way for truly dynamic orchestration solutions that surpass the limitations of traditional approaches.

2.2. Generative AI for Adaptive Automation

Generative AI has introduced a paradigm shift in software development, enabling the automated creation of functional code from high-level specifications or natural-language inputs [9]. Large language models (LLMs), trained on vast corpora of text and code, are now capable of not only completing small snippets but also devising entire programme structures with limited human intervention. Traditionally, code generation has relied on domain-specific languages or templates to automate routine tasks; however, such meth-

ods often lack flexibility when requirements evolve [10]. By contrast, LLM-based code generation leverages a context-rich understanding of user prompts, allowing dynamic composition of orchestration logic that can integrate multiple devices, protocols, or services. For instance, a user might state, “Whenever I start watching a movie, dim the living-room lights to 30% and close the curtains,” prompting the model to produce an automation script for these devices. This real-time translation of high-level intentions into executable code reduces the need for specialised programming knowledge and shortens development cycles.

Despite these advantages, the reliability of AI-generated code remains a central concern. LLMs may produce syntactically correct but semantically flawed outputs, misinterpret user prompts, or neglect edge-case handling [11]. Various mitigation strategies have been proposed, such as thorough testing, integrating domain-specific constraints, or requiring validation before deployment. The high stakes in smart home contexts—where user safety and privacy are paramount—amplify the need for trustworthy, verifiable code. Accordingly, researchers are exploring ways to blend the creativity of generative AI with formal verification techniques, runtime monitoring, or human oversight to ensure that flexible code generation meets the rigour demanded by production environments [12].

Building on these capabilities, Retrieval-Augmented Generation (RAG) has emerged as a technique that combines generative models with external repositories of domain-specific knowledge. Rather than relying solely on a model’s internal parameters, it retrieves contextual information from specialised databases, such as vector-based indices of device drivers or usage logs, and feeds it into the generative process in real time [13]. This approach ensures that the outputs remain current and context-aware, particularly in dynamic IoT settings where updates to firmware, device APIs, or usage patterns can rapidly change the orchestration landscape. For example, if a new brand of smart thermostat requires a specific API call, the retrieval mechanism can supply those details so that the generated code seamlessly integrates with the novel device.

By augmenting LLMs with retrieval capabilities, RAG stands to overcome a common limitation of generative models: hallucinations or inaccurate outputs arising from outdated or incomplete training data. In smart home orchestration, real-time access to verified data can reduce errors and encourage reliable automation decisions. Nevertheless, the interplay of on-demand retrieval and code generation introduces additional challenges, such as increased computational overhead and the need to manage multiple data sources simultaneously [14,15]. Maintaining performance and consistency in large-scale environments thus necessitates careful architectural design, including efficient indexing schemes and caching strategies.

Taken together, generative AI and retrieval-augmented generation offer complementary paths to more flexible and robust smart home orchestration. While LLM-based code generation enhances adaptability and lowers barriers to automation, RAG ensures that the logic produced remains accurate, current, and aligned with external knowledge bases. In subsequent sections, we explore how these approaches, supported by a modular and dynamic framework, can overcome the limitations of traditional rule-based methods and deliver orchestration solutions that readily adapt to the ever-changing demands of modern IoT environments.

2.3. OSGi for Modular and Dynamic Services

Advances in generative AI and retrieval-augmented techniques are indicative of a new era of flexibility and adaptability. However, the underlying software architecture must be capable of evolving in real-time without compromising overall system stability. In this regard, the OSGi (Open Services Gateway initiative) framework has emerged as

a robust solution, facilitating the seamless deployment, update, and removal of software components—often referred to as “bundles”—within a Java-based environment. The OSGi framework is distinguished by its well-defined lifecycle model, which enables each bundle to be initiated, halted, or updated in an isolated manner. This prevents disruptive restarts or periods of system downtime when new services are introduced, or existing services are modified [2,16].

The fundamental principle of modularity underpinning OSGi reflects a design philosophy that treats each service or functional component as an encapsulated unit with clearly defined interfaces and dependencies [17]. This approach helps avoid complex interdependencies that can cause cascading failures in traditional monolithic systems, thus boosting overall system resilience. In the context of smart home orchestration, OSGi facilitates the concurrent operation of isolated services, including device discovery, automation logic, user interfaces, and data analytics, thus enabling developers to address specific tasks in a targeted manner [18]. If a newly generated service bundle—potentially created by an AI model—introduces unforeseen issues or fails to comply with expected constraints, it can be rolled back or isolated without affecting other components.

This level of granularity facilitates dynamic reconfiguration of the system in response to evolving user needs or situational changes. For instance, if a user installs a new type of sensor or wishes to integrate a newly discovered device driver, OSGi can load and activate the relevant bundle on the fly, thereby eliminating the need for a system-wide overhaul. Furthermore, the framework’s inherent modularity seamlessly integrates with AI-driven orchestration methods, which often necessitate frequent updates or patches to accommodate the sophisticated logic generated by large language models. This synergy not only streamlines the deployment of updates but also enhances the system’s ability to adapt in near-real time.

However, it is important to note that the flexibility offered by OSGi does come with design and management considerations. The introduction or removal of service bundles must be carefully synchronised to avoid versioning conflicts, unmet dependencies, or security vulnerabilities, especially when code is generated or adapted autonomously. Notwithstanding the potential challenges, OSGi is distinguished as a proven solution for real-time, modular service management in smart home and IoT ecosystems. By integrating the framework’s robust architecture with emerging AI techniques, modern orchestration systems have the potential to transition from static, rule-based workflows to deliver adaptive, highly reliable smart home environments.

2.4. Open Challenges

Although the convergence of generative AI, retrieval-augmented techniques, and modular service frameworks such as OSGi brings us closer to seamlessly orchestrated smart home systems, several challenges remain unresolved. A primary concern lies in the heterogeneous landscape of IoT products, each often adhering to distinct protocols or standards that hinder smooth interoperability [19]. When AI-generated code is tasked with coordinating devices across disparate communication layers, it must rely on robust discovery mechanisms and robust abstraction layers to maintain a consistent user experience.

A second major hurdle involves scalability and performance. While retrieval-augmented generation provides context-rich outputs, frequent queries to external knowledge repositories can result in increased latency and resource consumption [15]. In settings with numerous devices and substantial data flows, efficient indexing and caching become pivotal for preserving real-time responsiveness. The potential need to generate or update orchestration logic on the fly amplifies the demand for efficient load balancing and

concurrency control so that occupants do not experience delays or disruptions in their daily routines.

Security and privacy considerations add another layer of complexity to these systems. Automatically generated code, particularly when introduced without direct human oversight, could inadvertently open vulnerabilities to malicious exploits or compromise sensitive data [20]. To mitigate such risks, modern smart home solutions must incorporate rigorous code validation, sandboxing methods, and robust encryption. Moreover, ethical and regulatory aspects surrounding the collection, processing, and storage of user data call for careful safeguards, ensuring occupants retain confidence in the system's trustworthiness.

An additional concern is code verification. Despite the remarkable strides made by large language models, no guarantee that automatically generated scripts will flawlessly address all edge cases or systematically account for exceptional scenarios. Researchers have, therefore, begun exploring formal verification methods to detect logical inconsistencies or security flaws before deployment [21]. However, these techniques can be resource-intensive and may limit the real-time adaptability that characterises AI-driven automation. Balancing the rigour of verification with the requirement for rapid, on-demand code generation remains a key research focus.

Finally, the successful adoption of AI-driven orchestration hinges on user acceptance and transparent decision-making. While generative AI can dramatically lower the barriers to automation, many individuals remain cautious about relinquishing control to autonomous algorithms. Designing intuitive interfaces that explain how decisions are made helps ensure that occupants feel empowered rather than marginalised by the technology. Such human-centred design principles become even more pertinent in scenarios involving vulnerable populations, such as older adults or individuals with disabilities.

Ultimately, addressing these technical, ethical, and social challenges demands multidisciplinary collaboration, encompassing fields as varied as software engineering, cybersecurity, human–computer interaction, and the social sciences. By facing these open questions head-on, the next generation of smart home systems can offer environments that are not only more adaptive and efficient but also aligned with the diverse needs and values of those who live in them.

3. System Overview

The orchestrator system is designed as a layered architecture that automates and manages smart home devices through event-driven mechanisms and intelligent processing. By dividing responsibilities among distinct layers, the system ensures modularity, extensibility, and robust functionality as each request transitions from a user's high-level idea to concrete actions on physical devices. This architectural design integrates:

- MQTT for device event messaging,
- Vector and relational databases for data storage and retrieval,
- The Apache Felix implementation of OSGi for modular service management, and
- A Large Language Model (LLM) for generating dynamic orchestration logic.

Figure 1 illustrates this architecture, highlighting how each component—from device registration through data processing to user interaction—contributes to a cohesive, adaptable smart home environment. When a new device comes online or a user issues a request, the system can promptly respond by deploying or updating services on the fly and executing the desired automation.

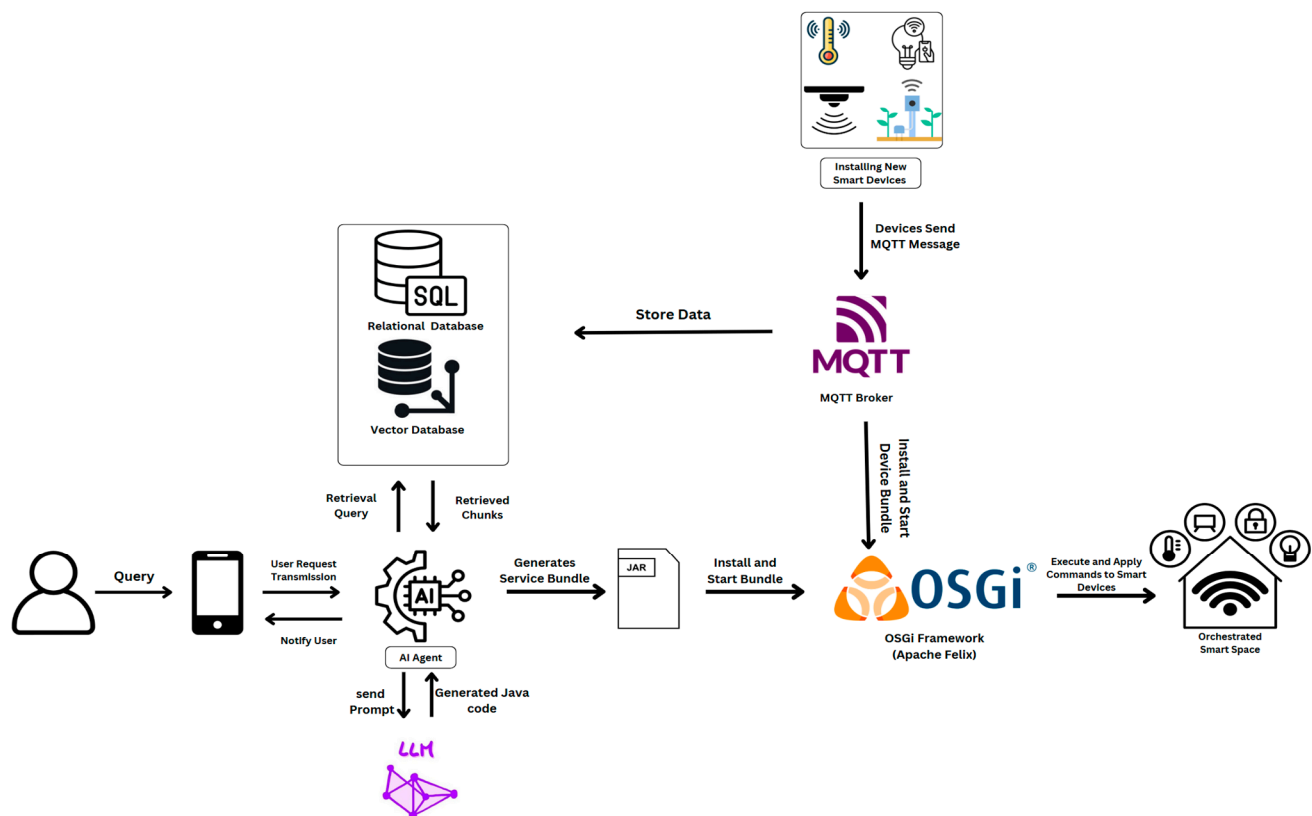


Figure 1. System architecture of the orchestrator; representing the integration of MQTT, databases, AI-driven code generation, and OSGi-based deployment for dynamic smart home automation.

The system's operation can be broadly divided into two key processes: the Device Installation Process (DIP) and the Service Bundle Generation Process (SBGP). The DIP addresses how new devices are onboarded, while the SBGP explains how user requirements are transformed into runnable code. Both processes rely on the interplay among the architectural layers described in Section 3.3, ensuring that each layer's output becomes the next layer's input.

3.1. Device Installation Process

When a new smart device enters the environment, the DIP automatically integrates it into the orchestrator following a well-defined sequence (see Figure 2). This process spans from the initial device announcement to the activation of its service within the OSGi framework, ensuring minimal manual intervention. The high-level steps include:

1. **Device Announcement:** Upon installation, the device broadcasts a registration message (for instance, via MQTT) indicating its identity and capabilities.
2. **Registration Processing:** The orchestrator records the device's information in the system's databases, mapping device attributes, capabilities, and configuration details for use in future automation scenarios.
3. **Service Activation:** After registration, the orchestrator triggers the installation of the device's service bundle into the OSGi framework. Once activated, the device's functionality is exposed as a service, ready for orchestration within the smart home.

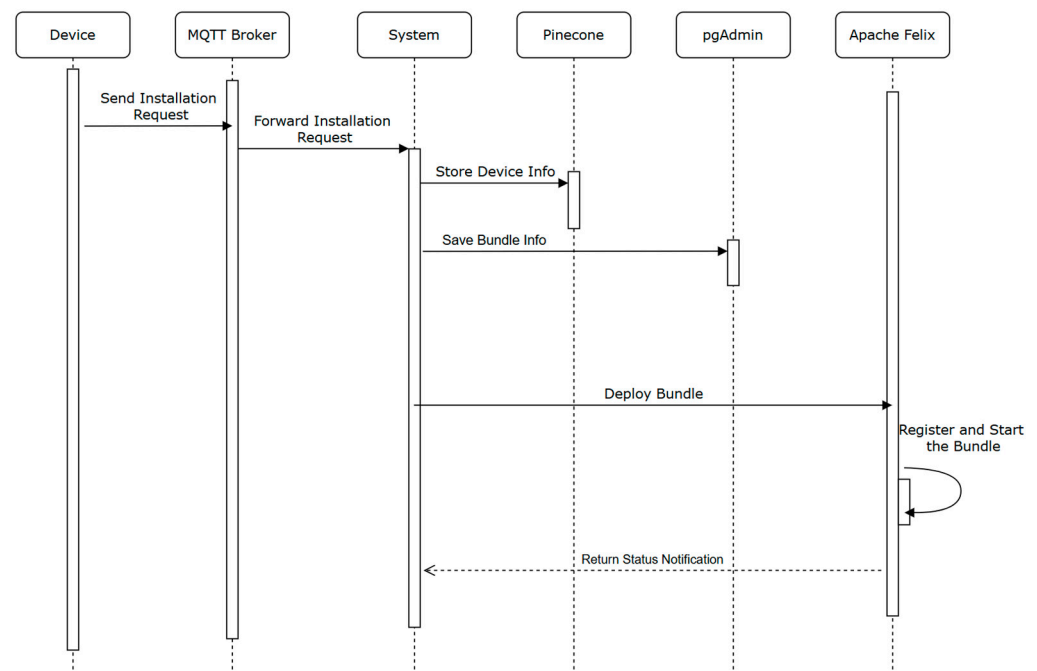


Figure 2. Device installation sequence diagram showing the discovery, registration, and service bundle activation process for newly introduced smart home devices within the orchestrator.

This sequence demonstrates how the system leverages MQTT for event-driven integration alongside OSGi for dynamic bundle deployment. Errors during installation, such as malformed device announcements, are logged, and the orchestrator gracefully handles partial failures by isolating problematic bundles, preserving the stability of the overall environment.

3.2. Service Bundle Generation Process

Building on the devices that have already been installed via the DIP, the Service Bundle Generation Process (SBGP) focuses on translating user requirements into runnable, on-demand code (see Figure 3). The process can be summarised as follows:

1. **User Requirement Input:** A user expresses a desired automation in natural language. For instance, “Adjust the living-room lamp brightness to 58%, turn on the speaker, and set its volume to 30”.
2. **Device Selection:** Based on the request, the orchestrator queries its databases—both vector-based for semantic matching and relational for structured lookups—to identify devices capable of fulfilling the tasks (e.g., lights for brightness control, and speakers for audio management).
3. **Automatic Code Generation:** Using the user’s command and device details, the orchestrator formulates a structured prompt for an LLM, which generates Java code tailored to orchestrate the selected devices. This code adheres to OSGi conventions (e.g., Declarative Services annotations) so that it can be readily deployed as a service bundle.
4. **Deployment and Execution:** The new bundle is compiled and installed via the OSGi framework, becoming an active component in the system. Once running, it executes the specified automation logic. In more advanced scenarios, the orchestrator may decide whether to keep the bundle active or uninstall it after use, depending on resource management policies.

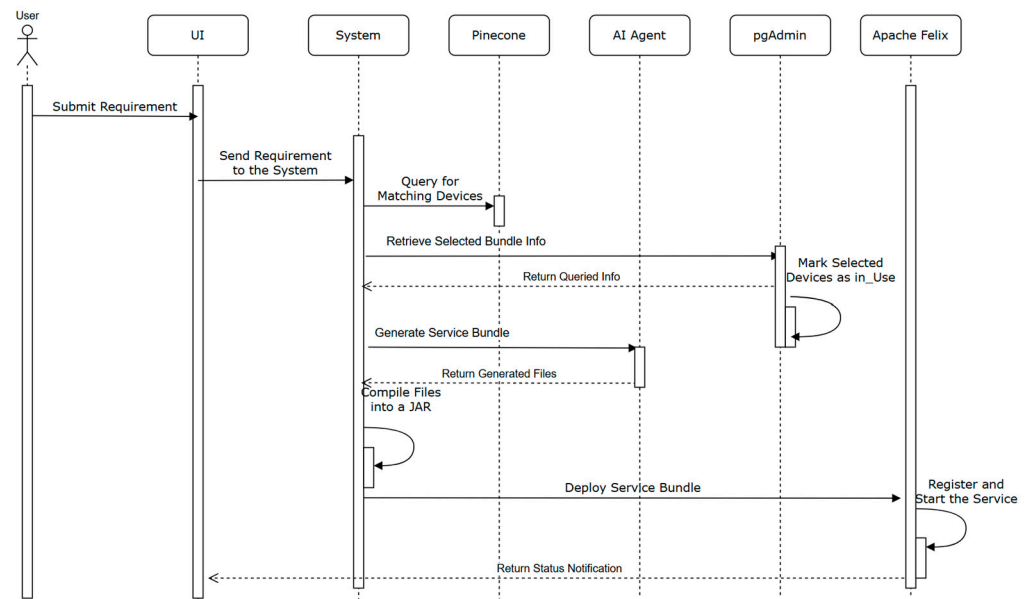


Figure 3. Service bundle generation process from natural language user input to automatic code generation, compilation, deployment, and activation within the OSGi framework.

Through this process, user requests are swiftly transformed into service bundles that run in near real-time, illustrating the synergy between generative AI and dynamic, modular frameworks.

3.3. Architectural Design and Components

Underpinning the DIP and SBGP is a four-layered architecture, each layer responsible for discrete tasks yet tightly interlinked to maintain overall coherence (see Figure 1 for an overarching view). In this subsection, we detail how data and control flow through the following layers: (1) the Device Registration Layer, (2) the Data Storage and Integration Layer, (3) the Control Layer, and (4) the Generative AI Integration Layer. By separating concerns into these layers, the system remains both extensible, which can integrate new devices and services and resilient, and able to cope with partial failures and evolving user demands.

3.3.1. Device Registration Layer

This layer oversees the onboarding of new devices from the moment they announce their presence via MQTT to the point where their service bundles become operational in OSGi. Key stages include:

1. **MQTT Message Publishing:** for uniform integration, device manufacturers must adhere to a predefined format for the device's announcement message. Upon first connecting or being installed, a smart device publishes a JSON-formatted message to a designated MQTT topic, such as *discovery/#*, where the wildcard # ensures the orchestrator captures all discovery messages. This registration message contains the key variables detailed in Table 1 [22].
2. **Message Consumption:** The orchestrator's MQTT client (running as a Python component) is continuously subscribed to the discovery topic to listen for new device messages. When a registration message is received, the MQTT client parses the JSON payload to extract the device information. The successful reception and parsing of the message mark the device as discovered by the system.
3. **Data Segregation and Storage:** Once the device data are parsed, the system stores it in two separate databases for different purposes. First, the device's descriptive

attributes (ID, name, type, capabilities, location) are vectorised into an embedding using Sentence Transformer techniques and sent to the vector database [23,24]. Storing this information in the vector database enables semantic queries—for example, finding devices that match a capability or location described in a user’s request (even if the request wording is not an exact literal match to the stored data). Each device’s embedding is stored with an ID and metadata. Second, the orchestrator stores configuration-specific details in a relational database. This includes the device’s configuration URL (for the bundle), the OSGi interface package and class that the device implements, method signatures offered by the device, and an entry in a device usage or similar table to track the device’s availability. Using a structured database for this information allows the orchestrator to perform fast lookups.

4. **Service Registration via OSGi:** After storing the device data, the orchestrator moves to integrate the device’s service bundle into the runtime. The Python MQTT handler invokes the Java controller by issuing a REST call (HTTP POST) with the device’s bundle URL. The controller, which leverages the OSGi framework, handles the bundle deployment. It uses an OSGi API call to fetch the JAR from the given URL, install it into the Apache Felix OSGi container, and start the bundle. This process is robust: the controller first verifies the bundle file can be retrieved and is not already installed, then performs the installation. Once the bundle is started, the device’s code (which implements certain service interfaces) is activated within the OSGi environment. During startup, the bundle typically registers its services with the OSGi Service Registry. This registration is often automated through Declarative Services: the bundle’s manifest file contains a pointer to an OSGI-INF component descriptor that declares the provided service interface and implementation. The OSGi Service Component Runtime then reads this and registers the new service. As a result, other components in the system can discover the device’s service via the registry. The process of bundle installation, activation, and service registration happens almost instantly and with minimal overhead. To summarise, the new device’s functionality becomes available to the orchestrator as a modular service without stopping or reconfiguring the running system.

Table 1. List of payload fields in the discovery message.

Field	Data Type	Description	Example
name	String	Commercial name of the device	<i>lightswitch3000</i>
ID	Number	Number ID of the device	<i>0252525</i>
manufacturer	String	Real name of the manufacturer	<i>Apple Inc.</i>
model	String	Model of the device	<i>J3000-p4-2024</i>
SW_version	String	Version of the device’s firmware	<i>3.4</i>
suggested_area	String	Recommended space area of use	<i>Livingroom</i>
main_feature	String	Main feature of the device	<i>Light dimming</i>
configuration_url	URI	Driver/Bundle’s URL	https://device.url.apple.com/j3000-p4-2024

Upon completion, the Device Registration Layer hands off a fully integrated device service to the rest of the system, enabling context-aware automation.

3.3.2. Data Storage and Integration Layer

This layer manages all data related to devices and their bundles, ensuring that the orchestrator can efficiently retrieve necessary information for decision-making and code generation. It consists of two main components:

- **Vector Database:** The orchestrator utilises a vector database to store embeddings of device information. By converting device attributes into embeddings, the system enables context-aware searches (a form of similarity search), which is crucial for matching user requests to relevant devices. The vectorization process concatenates key attributes of each device (such as the user or device ID, room, device name, and listed capabilities) into a single descriptive string. This string is then passed through a pre-trained Sentence Transformer model to produce a numeric embedding vector. The embedding captures semantic relationships. For example, devices in the “living room” with the capability “light” will have vectors positioned such that a query embedding for “living room lamp” finds them as nearest neighbours. Each resulting vector is stored with a unique ID and associated metadata (the raw attributes). When the orchestrator later needs to find devices for a user command, it embeds the query in the same way and asks the database for the closest matches, retrieving the IDs of the most relevant devices. This vector-based approach greatly enhances the system’s ability to select devices intelligently based on context rather than requiring exact keyword matches.
- **Relational Database:** Alongside the vector store, a relational database is used for maintaining detailed device and bundle information in tables. The schema includes tables such as *device_info*, where static info like device model, the interface package and class name, and available method names are stored; and *device_usage*, which is used to track the real-time usage status of each device and other dynamic states.

By separating concerns this layer provides a robust foundation for the orchestrator. It ensures both speed via vector search and accuracy via consistent relational data in the system’s operation.

3.3.3. Control Layer

The Control Layer is implemented as a Java-based controller that sits at the heart of the OSGi runtime, managing the lifecycle of all device service bundles. Its primary role is to act as the orchestrator’s deployer and manager for OSGi bundles, ensuring that new devices and newly generated orchestration bundles are smoothly integrated at runtime. The Control Layer leverages the modular nature of OSGi to achieve dynamic updates without shutting down the system [25,26]. Key functionalities of this layer include:

1. **Dynamic Bundle Management:** The controller can install, initiate, halt, and update OSGi bundles in real-time. When triggered, whether by the device registration process or by the code generation component, it employs OSGi framework APIs to install a bundle JAR into the Apache Felix container and initiate its *BundleActivator*, if present, or DS components. This dynamic lifecycle management ensures that as soon as a new device’s bundle or a new orchestration bundle is available, the controller brings it online. The controller then validates the existence and correctness of bundle files, ensuring the JAR is accessible via the provided URL, and subsequently calls the relevant OSGi methods to install and initiate the bundle. This modular approach facilitates the addition of new functionalities without necessitating the restart of the entire application.
2. **System Coordination:** The controller functions as an intermediary between the Device Registration layer and the AI Integration layer. While it does not process MQTT messages or user requests, it ensures that the results of those processes, i.e., device bundles to install or newly generated orchestration bundles, are realised in the running system. In summary, the controller assimilates the output of the preceding layers (a bundle to deploy) into the active system. Following the installation process, control is transferred to the OSGi framework’s service registry and lifecycle management for

the designated bundle. This delineation of responsibilities ensures that the Python component and the AI components do not need to directly manipulate OSGi internals; rather, they are delegated to the Control Layer.

3. **Scalability and Modularity:** The control layer contributes to the system's scalability by leveraging OSGi's inherent modular design. Each device or orchestration constitutes a self-contained bundle; the controller can add any number of such bundles at runtime if the resources permit. This design facilitates the straightforward extension of the system to hundreds of devices or numerous concurrent orchestration services. Additionally, the updating or removal of a device's functionality is equally modular: the controller can stop and uninstall bundles if needed, isolating any impact. The utilisation of OSGi ensures that bundles interact exclusively through well-defined services and that unloading a bundle results in the clean release of its resources. This layer, therefore, contributes to the maintenance and flexibility of the system by overseeing the dynamic composition of the application.
4. **Error Handling and Logging:** Robust error handling is an integral feature of the controller, ensuring the reliability and debuggability of the system. Significant events are logged by the controller (e.g., "Device bundle X installed successfully", or "Error: Bundle Y not found at URL..."). In the event of a bundle installation failing (for instance, due to a missing file or a malformed JAR), the controller catches the exception and records an error log. This logging and feedback mechanism is essential for maintenance and for the orchestrator to handle fallback behaviours.

3.3.4. Generative AI Integration Layer

The Generative AI Integration Layer is the core intelligence of the orchestrator system. It is responsible for transforming user requirements (expressed in natural language) into an executable form (OSGi service bundles) and making high-level decisions during orchestration. This layer combines techniques from natural language processing, semantic reasoning, and code generation. Its design is tightly coupled with OSGi's Declarative Services to ensure that any AI-generated components integrate seamlessly with the rest of the system. The AI Integration Layer can be broken down into several key components and steps:

1. **Natural Language Processing and Context Building:** The process commences with the comprehension of the user's request. The orchestrator employs fundamental NLP methodologies to parse the input text and extract the actionable information: the user's desired actions, the involved device or device type, and any specific parameters (e.g., location or values). In the provided example, "adjust the living room lamp brightness to 58% and turn on the speaker, setting volume to 30", the parser identifies two primary actions (adjust brightness, turn on and set volume) and their targets (a lamp in the living room and the speaker). To accurately map these to actual devices, the orchestrator then performs contextual reasoning using the data from the preceding layers. It embeds the parsed request (or key terms from it) into a vector and queries the vector database to find matching devices that are (a) in the living room and (b) capable of brightness adjustment (for the lamp part) or audio output (for the speaker part). The orchestrator then verifies the existence of these devices in the relational database, ensuring they are not already in use. This process ensures that the selected devices can fulfil the request and that assigning them will not conflict with other ongoing tasks. This step, which is integral to the system's functionality, translates the user's command into a set of concrete device references, thereby ensuring that the system's state can accommodate the request.
2. **Declarative Services in OSGi:** The orchestrator utilises OSGi Declarative Services (DS) as the foundational framework for all service components, encompassing those that

are generated on an as-needed basis. The use of Declarative Services enables bundle developers (or in this case, the AI system) to specify components, their provided services, and their dependencies via annotations and XML descriptors, thus obviating the need for imperative code. In the context of the orchestrator, when the LLM generates a new Java class for the orchestration logic, it uses DS annotations to declare, for example, that the class is a `@Component` (so it will be treated as an OSGi service component), that it requires certain services (e.g., a `LightService` and `SpeakerService`) via `@Reference` fields, and what method should run on activation (`@Activate`). The advantage of this approach is that when the bundle is deployed, the OSGi framework automatically handles service binding: the framework will read the bundle's metadata and see that the new component needs, say, an implementation of `LightService`. As the lamp's bundle has registered a `LightService` in the registry, the framework will inject that service into the new component.

Figure 4 illustrates the OSGi service registry and how declarative services interact within the framework [21]. It shows conceptually how services are published by bundles and then consumed by other components through the registry lookup, all managed by the OSGi container. By using DS, the orchestrator avoids manual lookup of services and ensures loose coupling between generated code and device implementations. Moreover, DS provides lifecycle management: if a service becomes unavailable, the component can be deactivated, etc., which adds to system robustness. Once the new orchestration bundle is started, its services (if any are provided) or at least its component existence is also registered. In most cases, though, the generated bundle is primarily a consumer of device services. As soon as it activates, the orchestrator's job for this request is completed. Figure 5 shows an example from the runtime: after deployment, device bundles (and by extension any orchestration bundles) appear in the Apache Felix console's service registry listing, confirming that they have been successfully registered and are ready to use.

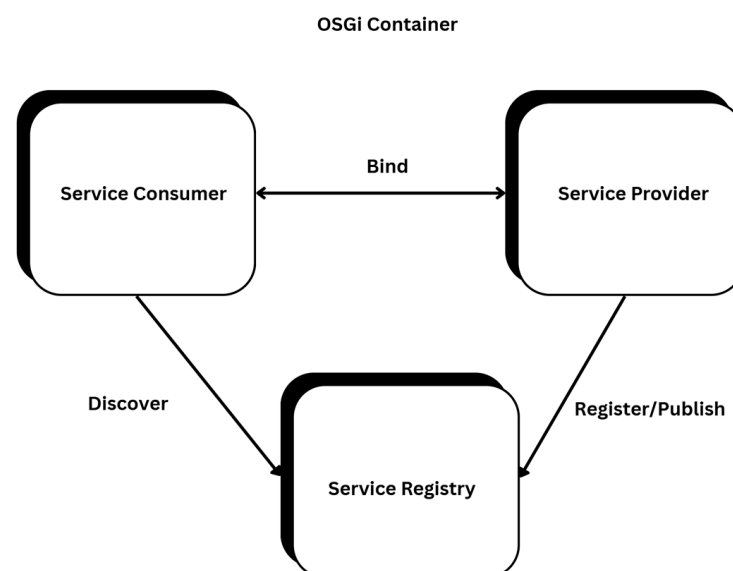


Figure 4. Interaction between service consumers, service providers, and the service registry within an OSGi framework, based on the model proposed in [21]. The service provider registers/publishes services, the consumer discovers them, and binding is handled dynamically by the registry.

```
scr:list
com.example.device.services.PhilipsHueLight in bundle 37 (com.example.device.services.PhilipsHueLight:1.0.0.202412091257)
  enabled, 1 instance.
  Id: 0, State:ACTIVE
homedevice.services.SmartSpeaker in bundle 69 (homedevice.services.SmartSpeaker:1.0.0.202412131133) enabled, 1 instance.
  Id: 1, State:ACTIVE
bedroom.device.services.SmartHeater in bundle 83 (bedroom.device.services.SmartHeater:1.0.0.202412181422) enabled, 1 instance.
  Id: 2, State:ACTIVE
garage.device.services.GarageDoor in bundle 87 (garage.device.services.GarageDoor:1.0.0.202412181512) enabled, 1 instance.
  Id: 6, State:ACTIVE
```

Figure 5. Apache Felix console output displaying device bundles registered as active services, confirming successful dynamic integration through the orchestrator.

3. **Automated Code Generation and Deployment:** A central innovation of this system is the dynamic creation of OSGi-compatible code via an AI model. After determining which devices and services are involved in a user request and gathering their interface details, the orchestrator formulates a structured prompt for the LLM (OpenAI's model) [27]. This prompt typically includes a summary of the user's request, a description of each relevant device service, and what the final code should accomplish. When the LLM returns the Java code, the code consists of a class implementing the desired logic in an *@Activate* method, with fields for each required device service (annotated with *@Reference* and setter methods for dependency injection). It also includes any helper methods corresponding to device action, i.e., an *execute* method or directly the *@Activate* method calls *lightService.setBrightness(58)*, *speakerService.turnOn()*, and *speakerService.setVolume(30)*. An important aspect is that the LLM has been guided to follow a template that matches the system's requirements, so the output code is likely to be correct without manual editing. Once the code is ready, the system programmatically compiles it. It uses the Java compiler with a runtime classpath that includes all required dependencies: notably, the JARs of the device bundles (to resolve *LightService*, *SpeakerService*, etc.), plus the OSGi core and DS libraries. The orchestrator generates the compilation command and executes it, producing a .class file for the new component [28,29].

Next, the orchestrator assembles the OSGi bundle: it creates a JAR file that contains this compiled class, and it adds the MANIFEST.MF file. The manifest contains entries such as *Bundle-SymbolicName*, *Bundle-Version*, and *Import*. It also includes the *Service-Component* entry, which points to the automatically generated component XML file in OSGI-INF (this XML lists the component and its implemented interfaces and references, as derived from the annotations in code). With these pieces in place, the orchestrator has a complete OSGi bundle JAR ready to go. Deployment is then handed off to the Control Layer: the orchestrator calls the controller to install this new bundle into Apache Felix. The controller installs and immediately starts the bundle, which triggers the OSGi SCR to activate the component. The generated code's *@Activate* method runs, executing the automation logic. If all goes well, the user's requested actions are performed on the target devices. The entire deployment happens, typically within the span of a few seconds after the user's input. After execution, the new bundle can remain in the system to listen for any events or simply be idle. In a more advanced implementation, one might uninstall the bundle after use to save resources, but that is a design choice; in this thesis implementation, bundles remain deployed, and their presence is harmless as they only act when triggered by their activation and then effectively go dormant.

4. **Contextual Reasoning and Decision-Making:** Beyond simply executing commands, the orchestrator exhibits a degree of intelligence in decision-making to handle edge cases and optimise performance. Contextual reasoning refers to the system's awareness of the current context of the smart home and using it to inform actions. For

example, the orchestrator keeps track of devices that are currently busy or recently were. If a user requests an action and the optimal device is currently occupied with another task, the orchestrator might choose an alternative device, if available, or queue the request rather than interfering or causing a conflict. Additionally, the system is aware of situations where a request cannot be fulfilled, for instance, if the user asks to dim the lights, but there are no dimmable lights in the specified room. In such cases, instead of failing silently, the orchestrator provides graceful handling: it can reply to the user indicating that no suitable device was found. This kind of feedback ensures transparency and improves user trust in the system.

4. Test Scenarios and Validation Results

To evaluate the proposed orchestrator system, we devised a series of test scenarios intended to validate three principal objectives:

1. Functional Correctness: Does the system accurately interpret user instructions and orchestrate device actions as requested?
2. Scalability and Extensibility: Can the system accommodate a growing number of devices and services under diverse conditions?
3. Adaptability and Intelligence: To what extent can the AI-driven component modify logic generation in response to varied user requirements and real-time events?

Each scenario addresses one or more of these objectives, offering comprehensive coverage of the system's capabilities. In total, we implemented three core validation scenarios:

- Device Installation,
- Single-Device Service Bundle Generation,
- Multi-Device Automation Execution.
- The following subsections detail the setup, methodology, and key observations for each scenario. By comparing the system's actual outcomes against the expected behaviour, we demonstrate how the orchestrator meets traditional smart home automation requirements.

4.1. Device Installation

This first test scenario evaluates how effectively and accurately the system incorporates a newly added device into the smart home ecosystem. It specifically verifies the extensibility requirement, showing that the orchestrator can detect, register, and activate additional devices at runtime without significant manual intervention.

In this trial, a simulated Smart Temperature device was introduced into the environment. To emulate an actual deployment, an MQTT client published a discovery message containing the device's identity, capabilities, and configuration details. Once this message was transmitted, the orchestrator's console logs confirmed proper receipt and parsing, as illustrated in Figure 6, indicating the system's ability to handle heterogeneous device announcements without requiring user-driven configuration steps.

After processing the discovery message, the orchestrator proceeded to install and start the corresponding OSGi bundle for the Smart Temperature device in the Apache Felix runtime. As shown in Figure 7, the bundle was installed and became Active, signifying that the device's functionality was now available to other components. Simultaneously, the device service was registered in the OSGi Service Registry, and verified through the *scr:list* command. Figure 8 confirms the presence of the service entry, ensuring that any orchestration logic, whether generated by AI or initiated by the user, can locate and interact with the Smart Temperature bundle.

```

Message received
discovery/lighting: b'{"id": "TemperatureSensor_401",\n "model": "TempSense T100",\n "device_name": "Room Temperature Sensor",\n "device_type": "TemperatureSensor",\n "manufacturer": "ClimateTech",\n "supported_protocol": "MQTT",\n "capabilities": "temperature monitoring",\n "user_id": "User_6",\n "room": "Living Room",\n "configuration_url": "TemperatureSensor.jar",\n "bundle_name": "TemperatureSensor",\n "interface_package_name": "climate.device.services",\n "interface_name": "TemperatureSensorService",\n "methods": [\n {\n "name": "double getTemperature()",\n "description": "Gets the current room temperature"\n },\n {\n "name": "void setThreshold(double threshold)",\n "description": "Sets a temperature threshold"\n }\n ]\n}' Retained flag is: False
Payload is: {
  "id": "TemperatureSensor_401",
  "model": "TempSense T100",
  "device_name": "Room Temperature Sensor",
  "device_type": "TemperatureSensor",
  "manufacturer": "ClimateTech",
  "supported_protocol": "MQTT",
  "capabilities": "temperature monitoring",
  "user_id": "User_6",
  "room": "Living Room",
  "configuration_url": "TemperatureSensor.jar",
  "bundle_name": "TemperatureSensor",
  "interface_package_name": "climate.device.services",
  "interface_name": "TemperatureSensorService",
  "methods": [
    {
      "name": "double getTemperature()",
      "description": "Gets the current room temperature"
    },
    {
      "name": "void setThreshold(double threshold)",
      "description": "Sets a temperature threshold"
    }
  ]
}

Message arrived processing...
Pinecone: Received device info for ID User_6_TemperatureSensor_401
Device TemperatureSensor_401 already exists in Pinecone.
Pinecone: Updating device info for ID: User_6_TemperatureSensor_401
Successfully updated device information with ID TemperatureSensor_401 in Pinecone
Successfully inserted device with ID: User_6_TemperatureSensor_401
Controller response: Bundle installed and started successfully: C:/Users/negin/Downloads/Bundles-Test/TemperatureSensor.jar

```

Figure 6. MQTT discovery message received for a new Smart Temperature device, demonstrating real-time device onboarding via event-driven messaging.

```

22|Active | 1|Apache Felix Declarative Services (2.2.4)|2.2.4
23|Active | 1|Apache Felix Web Management Console (5.0.8)|5.0.8
24|Active | 1|org.osgi.org.osgi.service.component (1.5.1.202212101352)|1.5.1.202212101352
25|Active | 1|org.osgi.org.osgi.service.component.annotations (1.5.1.202212101352)|1.5.1.202212101352
26|Active | 1|org.osgi.org.osgi.service.http (1.2.2.202109301733)|1.2.2.202109301733
27|Active | 1|org.osgi.org.osgi.service.log (1.5.0.202007221806)|1.5.0.202007221806
28|Active | 1|org.osgi.org.osgi.service.serviceloader (1.0.0.201505202024)|1.0.0.201505202024
29|Active | 1|org.osgi.org.osgi.util.converter (1.0.9.202202082230)|1.0.9.202202082230
30|Active | 1|org.osgi.org.osgi.util.function (1.2.0.202109301733)|1.2.0.202109301733
31|Active | 1|org.osgi.org.osgi.util.promise (1.3.0.202212101352)|1.3.0.202212101352
32|Active | 1|org.osgi.org.osgi.util.pushstream (1.1.0.202212101352)|1.1.0.202212101352
33|Active | 1|SLF4J API Module (2.0.9)|2.0.9
34|Active | 1|SLF4J Simple Provider (2.0.12)|2.0.12
35|Active | 1|Controller (1.0.0.qualifier)|1.0.0.qualifier
37|Active | 1|com.example.device.services.PhilipsHueLight (1.0.0.2024120912)|1.0.0.2024120912
69|Active | 1|homedevice.services.SmartSpeaker (1.0.0.202412131133)|1.0.0.202412131133
83|Active | 1|bedroom.device.services.SmartHeater (1.0.0.202412181422)|1.0.0.202412181422
87|Active | 1|garage.device.services.GarageDoor (1.0.0.202412181512)|1.0.0.202412181512
92|Active | 1|LivingRoomAutomation (1.0.0.qualifier)|1.0.0.qualifier
93|Active | 1|HeaterControlComponent (1.0.0.qualifier)|1.0.0.qualifier

g! Request received: /controller/install
Config URL received: C:/Users/negin/Downloads/Bundles-Test/TemperatureSensor.jar
Successfully installed and started bundle: climate.device.services.TemperatureSensor
lb

```

Figure 7. Console output confirming the successful installation and activation of the device bundle (Smart Temperature) into the Apache Felix OSGi environment.

Taken together, this discovery and installation process demonstrates the orchestrator's ability to integrate new devices in real-time. It also avoids the need for a system-wide restart and preserves a high degree of modularity since each new device becomes available as an independent service. These results highlight how the proposed architecture addresses the extensibility criterion by managing device additions with minimal disruption to ongoing operations.

```

94|Active | 1|climate.device.services.TemperatureSensor (1.0.0.202412181527)|1.0.0.202412181527
g! scr:list
com.example.device.services.PhilipsHueLight in bundle 37 (com.example.device.services.PhilipsHueLight:1.0.0.202412091257)
) enabled, 1 instance.
  Id: 0, State:ACTIVE
homedevice.services.SmartSpeaker in bundle 69 (homedevice.services.SmartSpeaker:1.0.0.202412131133) enabled, 1 instance.
  Id: 1, State:ACTIVE
bedroom.device.services.SmartHeater in bundle 83 (bedroom.device.services.SmartHeater:1.0.0.202412181422) enabled, 1 ins
tance.
  Id: 2, State:ACTIVE
garage.device.services.GarageDoor in bundle 87 (garage.device.services.GarageDoor:1.0.0.202412181512) enabled, 1 instanc
e.
  Id: 3, State:ACTIVE
LivingRoomAutomation in bundle 92 (LivingRoomAutomation:1.0.0.qualifier) enabled, 1 instance.
  Id: 4, State:ACTIVE
HeaterControlComponent in bundle 93 (HeaterControlComponent:1.0.0.qualifier) enabled, 1 instance.
  Id: 5, State:ACTIVE
climate.device.services.TemperatureSensor in bundle 94 (climate.device.services.TemperatureSensor:1.0.0.202412181527) en
abled, 1 instance.
  Id: 6, State:ACTIVE
g!

```

Figure 8. Apache Felix *scr:list* command output verifying the active registration of the device bundle service (Smart Temperature) in the system's Service Registry.

4.2. Single-Device Service Bundle Generation

The second test scenario examines the system's capacity to generate and deploy an automation service for a single device in response to a user request expressed in natural language. This scenario provides evidence of functional correctness, since it verifies whether the orchestrator can accurately interpret user commands and produce valid service bundles for corresponding device actions.

A user request was composed to reflect a simple, yet realistic use case: "Turn on the heater in the bedroom and set it to 25". Upon receiving this request, the orchestrator parsed the text and identified a suitable device, ensuring that the requirement was correctly matched to the Bedroom Heater. Figure 9 displays the user input alongside the system's confirmation of the selected device, thus validating the orchestrator's ability to translate natural language instructions into a concrete device mapping.

```

PROBLEMS  OUTPUT  TERMINAL  PORTS  AZURE  DEBUG CONSOLE
PS C:\Users\negin\OneDrive\Desktop\Master Thesis\RAG_System> python -u "c:\Users\negin\OneDrive\Desktop\Master Thesis\RAG_System\main.py"
Pinecone initialized successfully.
C:\Users\negin\AppData\Local\Programs\Python\Python312\Lib\site-packages\transformers\tokenization_utils_base.py:1601: FutureWarning: 'clean_up_tokenization_spaces'
s' was not set. It will be set to 'True' by default. This behavior will be deprecated in transformers v4.45, and will be then set to 'False' by default. For more d
etails check this issue: https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
Pinecone index connected: smart-devices
Enter user requirement (or type 'exit' to quit): Connected successfully to MQTT broker.
turn on the heater in the bedroom and set to 25
Devices chosen for user requirement:
Device ID: User_1_SmartHeater_105, Name: Bedroom Heater, Room: Bedroom, Capabilities: on/off, temperature control

```

Figure 9. User input and corresponding device selection process for a single-device automation scenario, illustrating natural language understanding and semantic device matching.

Once the correct device was located, the orchestrator prompted the AI component to generate the necessary Java code for an OSGi-compliant bundle. The system provided the relevant device interface details to the large language model, which returned a class implementation that invoked the heater's methods to turn it on and set the desired temperature. The orchestrator subsequently compiled and packaged this class into a bundle, following the OSGi standards required by the Apache Felix runtime. Figure 10 documents the code generation process, confirming that the service bundle was installed and started successfully, while Figures 11–13 present the resulting Java code, manifest, and component descriptor.

```

Java code saved to C:\Users\negin\Downloads\HeaterControlComponent_20adafc8\HeaterControlComponent.java
Compiled C:\Users\negin\Downloads\HeaterControlComponent_20adafc8\HeaterControlComponent.java successfully.
Classpath used: C:\Users\negin\Downloads\org.osgi.framework-1.8.0.jar;C:\Users\negin\Downloads\felix-framework-7.0.5\bundle\org.osgi.service.component.annotations-1.5.1.jar;C:\Users\negin\Downloads\Bundles-Test\SmartHeater.jar
Manifest file has been created at C:\Users\negin\Downloads\HeaterControlComponent_20adafc8\META-INF\MANIFEST.MF
DS descriptor file has been created at C:\Users\negin\Downloads\HeaterControlComponent_20adafc8\OSGI-INF\HeaterControlComponent.xml
added manifest
adding: HeaterControlComponent.java(in = 761) (out= 267)(deflated 64%)
ignoring entry META-INF/
ignoring entry META-INF/MANIFEST.MF
adding: OSGI-INF/(in = 0) (out= 0)(stored 0%)
adding: OSGI-INF/HeaterControlComponent.xml(in = 481) (out= 245)(deflated 49%)
adding: org/(in = 0) (out= 0)(stored 0%)
adding: org/smarthome/(in = 0) (out= 0)(stored 0%)
adding: org/smarthome/heatercontrol/(in = 0) (out= 0)(stored 0%)
adding: org/smarthome/heatercontrol/HeaterControlComponent.class(in = 952) (out= 492)(deflated 48%)
Created JAR file at C:\Users\negin\Downloads\HeaterControlComponent_f1a0aae7.jar.
Controller response: Bundle installed and started successfully: C:\Users\negin\Downloads\HeaterControlComponent_f1a0aae7.jar
Enter user requirement (or type 'exit' to quit): 

```

Figure 10. Terminal logs documenting Java code generation, compilation, and bundle deployment for the Bedroom Heater service in the orchestrator.

```

1 package org.smarthome.heatercontrol;
2
3 import org.osgi.service.component.annotations.Activate;
4 import org.osgi.service.component.annotations.Component;
5 import org.osgi.service.component.annotations.Reference;
6 import bedroom.device.services.SmartHeaterService;
7
8 @Component(immediate = true)
9 public class HeaterControlComponent {
10
11     private SmartHeaterService smartHeaterService;
12
13     @Reference
14     public void setSmartHeaterService(SmartHeaterService smartHeaterService) {
15         this.smartHeaterService = smartHeaterService;
16     }
17
18     @Activate
19     public void activate() {
20         controlHeater();
21     }
22
23     public void controlHeater() {
24         smartHeaterService.turnOn();
25         smartHeaterService.setTemperature(25);
26     }
27 }

```

Figure 11. Generated Java code for the Bedroom Heater service bundle, showing the structure and OSGi-compliant annotations produced by the AI component.

```

1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: HeaterControlComponent
4 Bundle-SymbolicName: HeaterControlComponent
5 Bundle-Version: 1.0.0.qualifier
6 Bundle-RequiredExecutionEnvironment: JavaSE-17
7 Import-Package: org.osgi.framework, org.osgi.service.component.annotations, bedroom.device.services
8 Service-Component: OSGI-INF/HeaterControlComponent.xml
9 Bundle-ActivationPolicy: lazy

```

Figure 12. Manifest file generated for the Bedroom Heater bundle, detailing the imported packages and bundle metadata required for successful OSGi deployment.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0"
3     name="HeaterControlComponent">
4
5     <implementation class="org.smarthome.heatercontrol.HeaterControlComponent"/>
6     <reference name="smartHeaterService"
7         interface="bedroom.device.services.SmartHeaterService"
8         cardinality="1..1"
9         policy="static"
10        bind="setSmartHeaterService"/>
11 </scr:component>
12

```

Figure 13. Component XML descriptor for the Bedroom Heater service, specifying the service dependencies and activation logic according to OSGi Declarative Services standards.

To verify that the heater responded to the new bundle, the logs in Figure 14 confirmed the bundle's activation in Apache Felix. The orchestrator's console reflected the correct execution of the user's instruction, and Figure 15 lists the newly registered service in the OSGi Service Registry. Lastly, Figure 16 illustrates the bundle's Active status in the Apache Felix web console, confirming that the system effectively carried out the requested automation.

```

g! Request received: /controller/install
Config URL received: C:\Users\negin\Downloads\HeaterControlComponent_f1a0aae7.jar
Smart Heater is turned on.
Smart Heater temperature set to: 25
Successfully installed and started bundle: HeaterControlComponent

```

Figure 14. Runtime activation logs in Apache Felix confirming that the Bedroom Heater service bundle started correctly and executed the intended device operations.

```

Id: 3, State:ACTIVE
LivingRoomAutomation in bundle 92 (LivingRoomAutomation:1.0.0.qualifier) enabled, 1 instance.
Id: 4, State:ACTIVE
HeaterControlComponent in bundle 93 (HeaterControlComponent:1.0.0.qualifier) enabled, 1 instance.
Id: 5, State:ACTIVE

```

Figure 15. Service registration listing showing the Bedroom Heater bundle registered successfully in the OSGi Service Registry after activation.

Apache Felix Web Console Bundles

OSGi Status Web Console Log out

Bundle information: 43 bundles in total - all 43 bundles active

Id	Name	Version	Category	Status	Actions
94	climate.device.services.TemperatureSensor (climate.device.services.TemperatureSensor)	1.0.0.202412181527		Active	   
93	HeaterControlComponent (HeaterControlComponent)	1.0.0.qualifier		Active	   
	Symbolic Name: HeaterControlComponent Version: 1.0.0.qualifier Bundle Location: file:file:/C:/Users/negin/Downloads/HeaterControlComponent_f1a0aae7.jar Last Modification: Wed Jan 08 18:46:22 CET 2025 Start Level: 1 Exported Packages: --- Imported Packages: bedroom.device.services,version=1.0.0 from bedroom.device.services.SmartHeater (83) org.osgi.framework,version=1.10.0 from org.apache.felix.framework (0) org.osgi.service.component.annotations,version=1.5.1 from org.osgi.service.component.annotations (25) Manifest Headers: Bundle-ActivationPolicy: lazy Bundle-ManifestVersion: 2 Bundle-Name: HeaterControlComponent Bundle-RequiredExecutionEnvironment: JavaSE-17 Bundle-SymbolicName: HeaterControlComponent Bundle-Version: 1.0.0.qualifier Created-By: 17.0.10 (Oracle Corporation) Import-Package: org.osgi.framework, org.osgi.service.component.annotations, bedroom.device.services Manifest-Version: 1.0 Service-Component: OSGI-INF/HeaterControlComponent.xml Required Capabilities: Capability "org.osgi.service.component.annotations" with directives filter="(s) [provided by org.apache.felix.framework (0)] Used Services: Service #77 of type(s) [bedroom.device.services.SmartHeaterService]				
92	LivingRoomAutomation (LivingRoomAutomation)	1.0.0.qualifier		Active	   
87	garage.device.services.GarageDoor (garage.device.services.GarageDoor)	1.0.0.202412181512		Active	   
83	bedroom.device.services.SmartHeater (bedroom.device.services.SmartHeater)	1.0.0.202412181422		Active	   

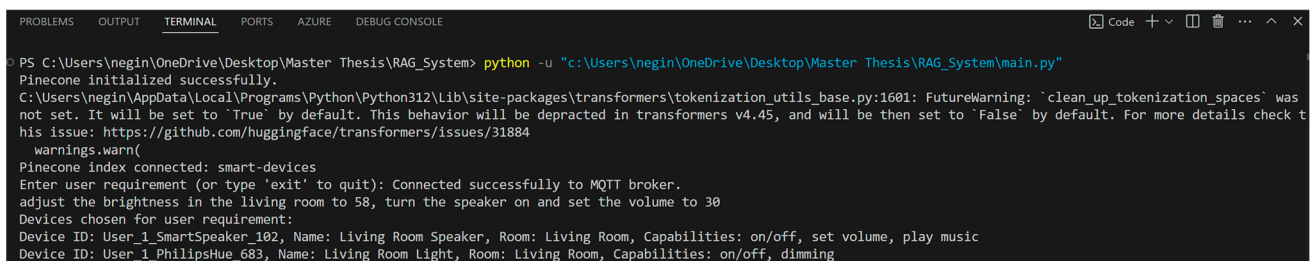
Figure 16. Apache Felix Web Console view highlighting the Bedroom Heater bundle in an active state with associated service components correctly registered.

Through these steps, the orchestrator demonstrated that it can transform a user's natural language command into a working OSGi service bundle with minimal human intervention. The success of this single-device test case highlights the orchestrator's functional correctness and offers a foundation for more complex scenarios involving multiple devices.

4.3. Multi-Device Automation Execution

The final test scenario involves a more complex automation request that requires coordination across multiple devices. This scenario is particularly relevant for validating the scalability and adaptability of the system, as it must parse a single user instruction that triggers distinct actions on two different devices.

In this case, the user requested: "Adjust the brightness in the living room to 58, turn the speaker on, and set the volume to 30". When the orchestrator received the command, it performed a semantic analysis to identify both a dimmable light and a speaker in the same room. As shown in Figure 17, the system logged the user input and confirmed that the Living Room Lamp and Living Room Speaker were the devices best suited to fulfil these requirements.



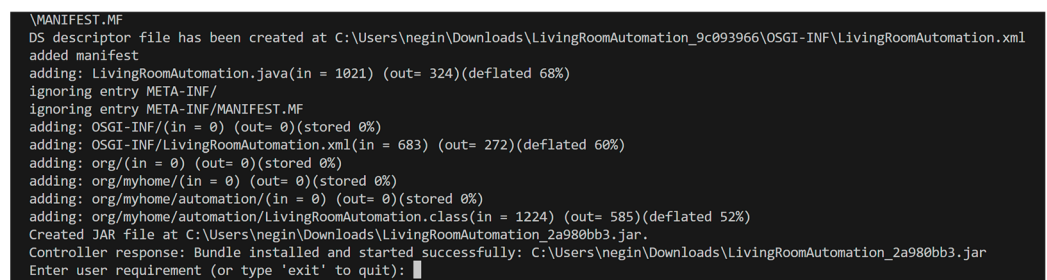
```

PROBLEMS OUTPUT TERMINAL PORTS AZURE DEBUG CONSOLE
PS C:\Users\negin\OneDrive\Desktop\Master Thesis\RAG_System> python -u "c:\Users\negin\OneDrive\Desktop\Master Thesis\RAG_System\main.py"
Pinecone initialized successfully.
C:\Users\negin\AppData\Local\Programs\Python\Python312\Lib\site-packages\transformers\tokenization_utils_base.py:1601: FutureWarning: `clean_up_tokenization_spaces` was not set. It will be set to `True` by default. This behavior will be deprecated in transformers v4.45, and will be then set to `False` by default. For more details check this issue: https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
Pinecone index connected: smart-devices
Enter user requirement (or type 'exit' to quit): Connected successfully to MQTT broker.
adjust the brightness in the living room to 58, turn the speaker on and set the volume to 30
Devices chosen for user requirement:
Device ID: User_1_SmartSpeaker_102, Name: Living Room Speaker, Room: Living Room, Capabilities: on/off, set volume, play music
Device ID: User_1_PhilipsHue_683, Name: Living Room Light, Room: Living Room, Capabilities: on/off, dimming

```

Figure 17. User input and multi-device selection process for a coordinated automation scenario involving a dimmable light and a smart speaker in the living room.

To execute the multi-device request, the orchestrator generated a new bundle named *LivingRoomAutomation*, which contained the coordination logic for both the lamp and the speaker. The AI component produced Java code specifying how the lamp's brightness level should be set to 58 and how the speaker's power and volume should be configured. Figure 18 shows the output logs confirming the successful creation and compilation of this bundle.



```

\MANIFEST.MF
DS descriptor file has been created at C:\Users\negin\Downloads\LivingRoomAutomation_9c093966\OSGI-INF\LivingRoomAutomation.xml
added manifest
adding: LivingRoomAutomation.java(in = 1021) (out= 324)(deflated 68%)
ignoring entry META-INF/
ignoring entry META-INF/MANIFEST.MF
adding: OSGI-INF/(in = 0) (out= 0)(stored 0%)
adding: OSGI-INF/LivingRoomAutomation.xml(in = 683) (out= 272)(deflated 60%)
adding: org/(in = 0) (out= 0)(stored 0%)
adding: org/myhome/(in = 0) (out= 0)(stored 0%)
adding: org/myhome/automation/(in = 0) (out= 0)(stored 0%)
adding: org/myhome/automation/LivingRoomAutomation.class(in = 1224) (out= 585)(deflated 52%)
Created JAR file at C:\Users\negin\Downloads\LivingRoomAutomation_2a980bb3.jar.
Controller response: Bundle installed and started successfully: C:\Users\negin\Downloads\LivingRoomAutomation_2a980bb3.jar
Enter user requirement (or type 'exit' to quit):

```

Figure 18. Terminal output confirming successful code generation, compilation, and deployment of the generated service bundle (*LivingRoomAutomation*) handling multiple devices.

Upon installation in the Apache Felix container, the orchestrator verified that the newly generated bundle included the relevant service references for both devices. Figures 19–21 display the Java code, manifest file, and component XML, confirming that the generated orchestration service recognised and used the devices' interfaces. When the *LivingRoomAutomation* component activated, the log messages in Figure 22 validated that

the tasks were carried out in the correct sequence: the brightness was set first, followed by the speaker's power-on event and volume adjustment.

```

1  package org.myhome.automation;
2
3  import org.osgi.service.component.annotations.Activate;
4  import org.osgi.service.component.annotations.Component;
5  import org.osgi.service.component.annotations.Reference;
6  import homedevic.services.SmartSpeakerService;
7  import com.example.device.services.LightService;
8
9  @Component(immediate = true)
10 public class LivingRoomAutomation {
11
12     private SmartSpeakerService smartSpeakerService;
13     private LightService lightService;
14
15     @Reference
16     public void setSmartSpeakerService(SmartSpeakerService smartSpeakerService) {
17         this.smartSpeakerService = smartSpeakerService;
18     }
19
20     @Reference
21     public void setLightService(LightService lightService) {
22         this.lightService = lightService;
23     }
24
25     @Activate
26     public void activate() {
27         adjustLivingRoom();
28     }
29
30     public void adjustLivingRoom() {
31         lightService.setBrightness(58);
32         smartSpeakerService.turnOn();
33         smartSpeakerService.setVolume(30);
34     }
35 }

```

Figure 19. Generated Java code for the service bundle, illustrating multi-device orchestration logic embedded within an OSGi Declarative Service component.

```

1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: LivingRoomAutomation
4  Bundle-SymbolicName: LivingRoomAutomation
5  Bundle-Version: 1.0.0.qualifier
6  Bundle-RequiredExecutionEnvironment: JavaSE-17
7  Import-Package: org.osgi.framework, org.osgi.service.component.annotations, com.example.device.services, homedevic.services
8  Service-Component: OSGI-INF/LivingRoomAutomation.xml
9  Bundle-ActivationPolicy: lazy

```

Figure 20. Manifest file created for the service bundle, documenting the necessary imports and bundle metadata for multi-device orchestration.

Finally, Figures 23 and 24 illustrate the new service's entry in the OSGi Service Registry and its Active status in the Apache Felix console. These confirmations show that the orchestrator handles multi-device scenarios without conflicts or execution delays. By coordinating the actions of separate devices within a single request, the system demonstrates the scalability and adaptability required for more advanced applications, where multiple devices often collaborate to provide a seamless user experience.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0"
3      name="LivingRoomAutomation">
4      <implementation class="org.myhome.automation.LivingRoomAutomation"/>
5      <reference name="smartSpeakerService"
6          interface="homedevice.services.SmartSpeakerService"
7          cardinality="1..1"
8          policy="static"
9          bind="setSmartSpeakerService"/>
10     <reference name="lightService"
11         interface="com.example.device.services.LightService"
12         cardinality="1..1"
13         policy="static"
14         bind="setLightService"/>
15 </scr:component>

```

Figure 21. Component XML descriptor for the generated service bundle, specifying service references for both the light and speaker components orchestrated in the user request.

```

g! Request received: /controller/install
Config URL received: C:\Users\negin\Downloads\LivingRoomAutomation_2a980bb3.jar
Philips Hue Light brightness set to level: 58
Smart Speaker is turned on.
Smart Speaker volume set to: 30
Successfully installed and started bundle: LivingRoomAutomation

```

Figure 22. Apache Felix runtime logs showing successful activation and execution of the generated service bundle's coordinated device actions.

```

LivingRoomAutomation in bundle 92 (LivingRoomAutomation:1.0.0.qualifier) enabled, 1 instance.
Id: 11, State:ACTIVE
q!

```

Figure 23. Service registration output in the OSGi Service Registry confirming the successful registration of the LivingRoomAutomation bundle and its service dependencies.

Apache Felix Web Console Bundles



OSGi Status Web Console							Log out
Bundle information: 43 bundles in total - all 43 bundles active							
Apply Filter Filter All		Reload Install/Update... Refresh Packages					
Id	Name	Version	Category	Status	Actions		
94	climate.device.services.TemperatureSensor (climate.device.services.TemperatureSensor)	1.0.0.202412181527		Active			
93	HeaterControlComponent (HeaterControlComponent)	1.0.0.qualifier		Active			
92	LivingRoomAutomation (LivingRoomAutomation)	1.0.0.qualifier		Active			
	Symbolic Name	LivingRoomAutomation					
	Version	1.0.0.qualifier					
	Bundle Location	file:file:/C:/Users/negin/Downloads/LivingRoomAutomation_2a980bb3.jar					
	Last Modification	Mon Dec 30 23:14:03 CET 2024					
	Start Level	1					
	Exported Packages	---					
	Imported Packages	com.example.device.services,version=1.0.0 from com.example.device.services.PhillipsHueLight (37) homedevice.services,version=1.0.0 from homedevice.services.SmartSpeaker (69) org.osgi.framework,version=1.10.0 from org.apache.felix.framework (0) org.osgi.service.component.annotations,version=1.5.1 from org.osgi.service.component.annotations (25)					
	Manifest Headers	Bundle-ActivationPolicy: lazy Bundle-ManifestVersion: 2 Bundle-Name: LivingRoomAutomation Bundle-RequiredExecutionEnvironment: JavaSE-17 Bundle-SymbolicName: LivingRoomAutomation Bundle-Version: 1.0.0.qualifier Created-By: 17.0.10 (Oracle Corporation) Import-Package: org.osgi.framework, org.osgi.service.component.annotations, com.example.device.services, homedevice.services Manifest-Version: 1.0 Service-Component: OSGI-INF/LivingRoomAutomation.xml					
	Required Capabilities	Capability "osgi.lee" with directives filter=(!(osgi.lee=JavaSE)(version=17.0.0)) (provided by org.apache.felix.framework (0))					
	Used Services	Service #76 of type(s) [homedevice.services.SmartSpeakerService] Service #75 of type(s) [com.example.device.services.LightService]					
87	garage.device.services.GarageDoor (garage.device.services.GarageDoor)	1.0.0.202412181512		Active			
83	bedroom.device.services.SmartHeater (bedroom.device.services.SmartHeater)	1.0.0.202412181422		Active			
69	homedevice.services.SmartSpeaker (homedevice.services.SmartSpeaker)	1.0.0.202412131133		Active			

Figure 24. Apache Felix Web Console display showing the LivingRoomAutomation bundle active with correct component binding and service exposure.

4.4. Quantitative Performance Evaluation

To complement the qualitative observations above, we introduce quantitative, numerical metrics characterising the orchestrator's behaviour during practical validation runs. Performance data were collected using 32 GB RAM, running Windows 10 and Apache Felix 7.0.5 as the OSGi container.

Measurement Methodology

- Context build time: from receiving user input to completion of prompt construction.
- Code generation time: duration of Java code generation via the LLM.
- Java compilation and packaging time: total time to compile Java code and package it into an OSGi JAR.
- Memory Consumption: peak resident-set-size (RSS) after context generation and after bundle activation, measured with *psutil*.
- Reliability: proportion of successful orchestrations across repeated runs.

All time measurements used Python’s high-resolution timer (*time.perf_counter()*), and memory was sampled via *psutil.Process().memory_info().rss*. Reliability statistics were computed over 20 automated runs using our *batch_test.py* harness: each run was marked successful only if the LLM generated bundle compiled, deployed in Apache Felix, and executed all intended device actions without exception. Table 2 shows the summary of results obtained.

Table 2. Orchestrator performance metrics: execution time and memory usage.

Evaluation Parameter	Mean $\pm$ SE	Description
Context build time	1.51 s $\pm$ 0.32 s	Prompt complexity affects variability.
Code generation time	3.81 s $\pm$ 0.44 s	Dominant LLM latency contributor.
Java compilation time	1.04 s $\pm$ 0.12 s	Compiles vary with interface count.
JAR packaging time	0.44 s $\pm$ 0.07 s	Generally stable across all cases.
Memory (post-context)	415.91 MB $\pm$ 4.85 MB	Higher when more devices are included.
Memory (post-activation)	416.40 MB $\pm$ 5.02 MB	Slightly increases with multi-device bundles.

As described in Table 3, reliability was measured by executing the complete orchestration pipeline in a fully automated loop. From 20 trials:

Table 3. Reliability metrics based on 20 executions: success rate and error margins.

Reliability Metric	Value
Success count	18/20
Reliability pp	90%
Standard error (SE)	$\pm 6.71\%$
95% CI margin of error	$\pm 13.15\%$

As shown in Figure 25, code generation accounts for 56.0% of the total pipeline time, while context build, compilation, and packaging contribute 22.2%, 15.3%, and 6.5%, respectively. This distribution underscores that LLM processing is the primary latency consumer, whereas compilation and JAR packaging introduce relatively minor overheads. Memory consumption remains stable around 416 MB between the end of prompt assembly and post-activation, demonstrating a consistent resource footprint suitable for edge gateway deployment. Over 20 test runs, the orchestrator achieved a 90% success rate (18/20) with a standard error of $\pm 6.7\%$ ($\pm 13.2\%$ at 95% confidence), indicating robust execution across repeated automated scenarios.

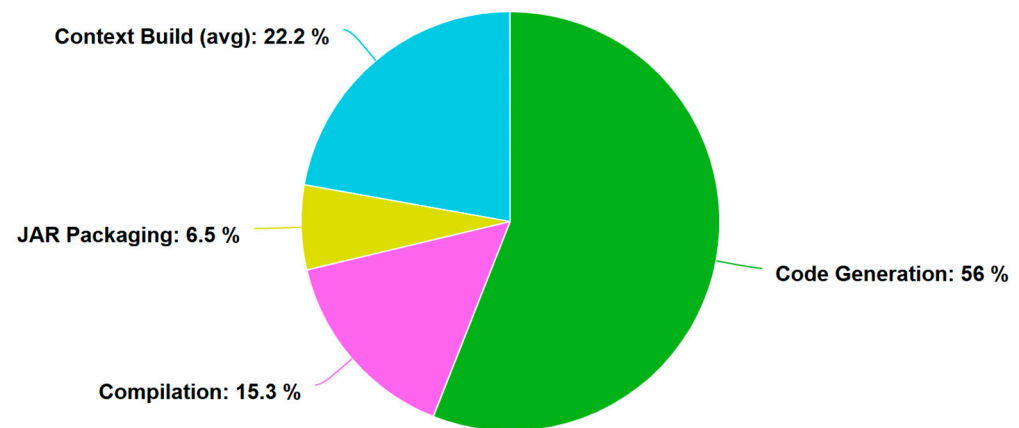


Figure 25. Proportional time distribution chart across orchestration phases including context building, code generation, compilation, and JAR packaging.

4.5. Model Specialisation and Its Impact on Generated Code Quality

While OpenAI’s LLMs, such as GPT-4, offer strong general-purpose code generation capabilities, our experience shows that domain specialisation is essential for reliable orchestration in smart home systems. Without any structured prompt templates or contextual guidance, the model’s raw outputs failed to compile or function in nearly all cases—demonstrating a near 100% failure rate. This confirms that generic LLMs, when used without domain awareness, are insufficient for generating compliant OSGi service bundles. In our setup, we define two categories of usage:

- **Base LLM:** GPT-4 used without any domain-specific prompting, examples, or contextual guidance. This reflects the “out-of-the-box” usage scenario.
- **Specialised LLM:** GPT-4 guided using structured prompt templates, enriched with project-specific examples (e.g., interface names, method descriptions, OSGi DS structure), simulating the effect of a fine-tuned model.

While we did not perform full fine-tuning, we evaluated the effect of simulated specialisation by feeding the model curated examples from our own orchestration codebase. This included:

1. Manufacturer-provided Java interface definitions and compiled bundles;
2. Existing OSGi Declarative Services (DS) annotations, XML descriptors, and manifest files;
3. Historical prompts and successful AI-generated service bundles.

To quantify the impact of this simulated specialisation, we ran a controlled experiment. GPT-4 was used to generate orchestration bundles for 20 distinct user requirements. The outputs were evaluated based on:

- **Compile-error rate:** Whether the generated Java code compiled on the first attempt.
- **Unsatisfied-reference rate:** The number of unresolved service bindings in Apache Felix (from SCR logs).
- **Manual edit time:** Time required by a developer to correct the code and successfully deploy the bundle.

These results, summarised in Table 4, highlight dramatic improvements. With structured domain prompts, the model reliably generated code that compiled, deployed, and operated with minimal or no manual correction. Specifically, the compile-error rate dropped from nearly 100% to 5%, and service reference errors (e.g., missing @Reference bindings or mismatched interface names) were almost entirely eliminated. Additionally, average

human correction time was reduced by over 80%, with many generated bundles requiring no intervention at all.

Table 4. Comparison between base and specialised LLM in code generation quality.

Metric	Base LLM	Specialised LLM	Improvements
Compile-error rate	~100%	5% (1/20)	~95 percentage point drop
Unsatisfied-reference rate	>1.0 errors/run	0.05 errors/run	Near elimination
Manual edit time (mean)	>30 s	5 s	~83% reduction

The improvement stems from several benefits introduced through specialisation. The model consistently used correct package names, OSGi annotations, and device interface references, which previously were common points of failure. It also maintained naming conventions and structure aligned with our DS component model, improving code clarity and compatibility. Importantly, the specialised prompts helped the model manage complex multi-device orchestration scenarios more effectively—reducing logical errors like out-of-order method calls or incorrect sequencing of device actions.

Therefore, even without retraining the model, simulating domain specialisation via tailored prompt templates and curated examples substantially improves both the quality and robustness of generated bundles. These results suggest that future work involving actual fine-tuning could further enhance performance, but that even prompt-level specialisation is highly effective and essential for deployment-grade orchestration.

4.6. Stress Testing and Scalability Evaluation

To assess the orchestrator’s performance under more realistic and demanding conditions, we conducted a stress test using 15 simulated smart home devices across different categories (smart lights, thermostats, motion sensors, speakers, and locks). This test aimed to evaluate how the system handles concurrent orchestration requests, manages device usage, and maintains performance under load. The test setup is detailed below, and the results are summarised in Table 5.

Table 5. Stress test results: latency, success rate, and memory usage.

Metric	Observed Value/Range
Avg. Orchestration Latency	4.2 s (sequential)–5.9 s (simultaneous)
Orchestration Success Rate	90% (9/10 successful executions)
Memory Usage (peak RSS)	448 MB
Device Contention Errors	1 occurrence (resolved on retry)

- Devices: 15 simulated devices introduced through MQTT discovery.
 - User Requirements: A total of 6 sequential orchestration requests followed by 4 simultaneous requests, each targeting different combinations of devices.
 - Execution Method: Sequential requests were executed one after the other. The simultaneous requests were run in parallel using Python’s *ThreadPoolExecutor*, simulating concurrent smart home events such as scheduled automations or overlapping voice commands
- Metrics collected:
- Orchestration Latency: Time from user input to bundle activation in Felix.
 - Success Rate: Number of orchestrations completed without error.
 - Memory Usage: Peak RAM usage during the orchestration cycle.

- Device Contention Handling: Number of orchestration attempts delayed or retried due to conflicting device usage.

During sequential execution, the orchestrator maintained a consistent response time of approximately 4.2 s per orchestration. When handling the 4 concurrent orchestration requests, latency slightly increased to an average of 5.9 s, primarily due to device selection conflicts and parallel bundle handling.

Of the 10 total orchestration attempts, 9 completed successfully. One failure occurred due to a device contention issue (i.e., two services attempting to control the same device simultaneously), which was automatically detected and retried using the orchestrator's internal resource tracking logic (DeviceUsageTracker). This component ensures safe, thread-aware management of device availability.

The system remained stable throughout the test, with memory usage peaking at around 448 MB, demonstrating suitability for deployment on lightweight home automation gateways. No unhandled exceptions, system crashes, or unstable service states were observed.

These results show that the orchestrator scales well in moderately complex environments, handling multiple active devices and overlapping automation requests with minimal performance impact and high reliability. The test confirms the system's ability to coordinate device usage efficiently and recover gracefully from temporary resource conflicts.

4.7. Model Validation and Future Direction for Failure Management

Because the orchestrator generates executable code that directly interacts with smart home devices, ensuring that this code is safe, correct, and functional is essential. While we do not yet employ formal verification techniques or automated test suites, the current system incorporates several manual validation mechanisms integrated into the deployment and monitoring process.

The generated Java bundles are organized into the following categories:

- Compiled and deployed directly into the Apache Felix runtime, which serves both as the development and execution environment.
- Manually verified using logs and OSGi SCR tools, such as *scr:list* and *scr:info*, to ensure successful bundle activation and service binding.
- Checked by developers for correctness, including structural validity of the Java class, proper annotations, and matching component XML descriptors.
- Monitored at runtime, where any failed activations or unsatisfied references are detected through Felix logs, and the corresponding orchestration process is halted.

This setup enables effective testing in small-scale experiments; it is not isolated from the core system and does not constitute a true sandbox. As a result, validation and failure management rely on manual oversight and cannot guarantee safety in larger, unattended deployments.

We acknowledge that this is a limitation of the current system and that scaling the orchestrator will require the introduction of more automated and formal validation strategies. In future iterations, we plan to:

- Introduce a dedicated sandbox environment, such as a separate Apache Felix instance, to safely test and validate generated bundles before integration.
- Integrate static analysis tools to detect code structure issues, annotation usage errors, and unresolved references before compilation.
- Explore automated testing pipelines, where test cases are generated and matched against expected orchestration outcomes.
- Investigate the use of formal verification techniques, including model checking and symbolic execution, to validate behavioural properties such as state reachability, sequencing correctness, and safety constraints.

Although these directions are not yet implemented, they are recognised as essential for advancing the system toward reliable, fully autonomous orchestration. Our current work provides a foundation for these capabilities by emphasising structured generation, logging, and controlled error handling. Future development will focus on formalising this process and enabling safe scalability through automation and validation frameworks.

5. Discussion

The evaluation of the orchestrator system confirms its alignment with the primary objectives of extensibility, scalability, and adaptability in smart home automation. By leveraging AI-driven code generation, a modular OSGi architecture, and dynamic service management, the system was shown to automate workflows based on real-time events and user requirements.

A key finding is the high level of scalability achieved through the orchestrator's design. The OSGi framework, along with the Apache Felix service registry, enabled the system to accommodate an expanding number of devices without noticeable performance degradation. During tests with multiple devices operating simultaneously, the orchestrator maintained swift response times and consistent system stability, which demonstrates its readiness for real-world deployment in diverse IoT environments. This outcome underscores the value of a modular approach that isolates services within discrete bundles and relies on lightweight, event-driven communication for orchestration.

Extensibility emerged as another notable strength of the system. Newly introduced devices were dynamically registered through MQTT announcements and installed as separate OSGi bundles at runtime. This arrangement allowed each device to appear as an independent service, discoverable by orchestration logic in real time. The process of declarative service registration via OSGi further streamlined the integration of device functionalities, thus reducing the need for manual configuration or extensive modifications to existing components. Moreover, the AI agent's architecture facilitates future enhancements, including more advanced natural language processing, which may accommodate increasingly complex user instructions.

The orchestrator also demonstrated adaptability by accepting natural language requests and translating them into executable Java-based service bundles without a full system restart. The system successfully deployed logic to perform tasks such as brightness adjustments and speaker volume controls across multiple devices, thereby validating its capacity to handle dynamic automation requirements. This adaptability is particularly advantageous in ever-evolving smart home environments, where user preferences and device capabilities frequently change.

Reliability was upheld throughout rigorous testing. The orchestrator consistently executed automation tasks without failures, even under conditions involving rapid device additions or unexpected user inputs. Its ability to gracefully handle device unavailability, while preserving overall stability, speaks to its robust design principles. The capacity to maintain system integrity despite continuous updates underscores the potential of this architecture to support mission-critical or comfort-oriented smart home scenarios.

In comparing this system with existing rule-based or partially automated frameworks, a salient aspect of novelty is the seamless integration of AI-driven code generation into a dynamic orchestration environment. Unlike conventional systems that rely on pre-programmed rules or offline code creation, the orchestrator presented here can autonomously generate and deploy service bundles at runtime in response to natural language requests. This real-time synergy between an LLM-based module and OSGi's modular services surpasses many prior solutions that require manual reconfiguration or cannot adapt rapidly to newly discovered devices or evolving user demands. Additionally, the two-

tiered data storage strategy (vector plus relational databases) further distinguishes this framework by enabling semantic searches that better capture user intent, thereby bridging the gap between unstructured requests and fully operational bundles.

While the system's multi-device orchestration was validated through a range of tests, there remain opportunities to enhance functionality and further differentiate the orchestrator from other state-of-the-art solutions. For instance, code generation relied on a general-purpose language model, and domain-specific LLMs trained on smart home or OSGi-related data could achieve higher accuracy and efficiency, particularly for multi-step, context-rich user requests. Additional improvements, including dynamic bundle updates, inter-bundle communication, and broader device compatibility, would help the orchestrator manage increasingly sophisticated workflows that span heterogeneous devices and networks.

By comparing the orchestrator's capabilities against similar approaches in the literature, this work demonstrates that fusing AI-driven code generation with modular, dynamic service management can advance the adaptability and intelligence of smart home systems. Nonetheless, ongoing refinements in model specialisation, system-level integration, and advanced coordination mechanisms are pivotal to fully realising a holistic and user-centred solution for the next generation of IoT-driven environments.

6. Conclusions

This work introduced a novel smart home orchestrator that integrates AI-driven code generation with dynamic, OSGi-based architecture, offering a flexible means of automating heterogeneous devices in real-time. By leveraging a large language model for on-demand service bundle creation, the system moves beyond traditional rule-based methods and demonstrates a high degree of adaptability to changing user preferences and device capabilities. Through multiple test scenarios, the orchestrator was shown to efficiently incorporate new devices, as well as scale to manage multiple services without performance degradation. Its modular design, enhanced by MQTT-based device registration and a vector-plus-relational data layer, underscores the tangible benefits of uniting advanced AI approaches with robust modular frameworks.

Nonetheless, several areas present themselves as opportunities for future research and development. While a general-purpose language model grants flexibility, it may not fully capture the nuances of specific domains or user contexts. Adopting a domain-specific LLM, trained on smart home or OSGi service patterns, could increase precision and reduce the need for manual oversight in code generation. Similarly, the system's reliance on discrete bundle installations for each automation change suggests that live updates or inter-bundle communication mechanisms could further enhance its responsiveness. Moreover, supporting device ecosystems beyond Java/OSGi through interoperable APIs or language-agnostic interfaces would make the orchestrator applicable to a broader range of IoT platforms. Finally, ensuring the correctness of automatically generated code highlights the importance of verification strategies, including static analysis or model checking, to detect logical errors or security vulnerabilities before deployment. Addressing these enhancements would empower the orchestrator to handle more complex and large-scale environments, ultimately steering the field toward a more comprehensive and user-centric vision of future smart homes.

Author Contributions: Conceptualization, N.J., M.V.-B. and I.P.; methodology, N.J. and M.V.-B.; software, N.J.; validation, N.J., M.V.-B. and L.E.-M.; formal analysis, N.J. and M.V.-B.; investigation, N.J., M.V.-B., L.E.-M., H.M., C.G.-V. and I.P.; resources, N.J.; data curation, N.J.; writing—original draft preparation, N.J., M.V.-B., C.G.-V., L.E.-M. and H.M.; writing—review and editing, M.V.-B., C.G.-V. and N.J.; visualisation, N.J.; supervision, M.V.-B. and C.G.-V.; project administration, I.P.,

M.V.-B. and C.G.-V.; funding acquisition, M.V.-B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analysed in this study. Data sharing does not apply to this article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Rajuroy, A.; Johnson Mary, B.; Liang, W. Personalized Smart Home Environment Management Using Context-Aware Reinforcement Learning. Available online: https://www.researchgate.net/publication/387820872_Personalized_Smart_Home_Environment_Management_Using_Context-Aware_Reinforcement_Learning (accessed on 1 April 2025).
2. Redondo, R.P.D.; Vilas, A.F.; Cabrer, M.R.; Arias, J.J.P.; Lopez, M.R. Enhancing Residential Gateways: OSGi Service Composition. *IEEE Trans. Consum. Electron.* **2007**, *53*, 87–95. [CrossRef]
3. Vermesan, O.; Bröring, A.; Tragos, E.; Serrano, M.; Bacciu, D.; Chessa, S.; Gallicchio, C.; Micheli, A.; Dragone, M.; Saffiotti, A.; et al. Internet of robotic things—converging sensing/actuating, hyperconnectivity, artificial intelligence and IoT platforms. In *Internet of Things: The Call of the Edge*; Vermesan, O., Bacquet, J., Eds.; CRC Press: Boca Raton, FL, USA, 2022; Volume 4, pp. 91–128.
4. Anik, S.M.H.; Gao, R.; Zhong, H.; Wang, X.; Meng, N. Automation Configuration in Smart Home Systems: Challenges and Opportunities. *arXiv* **2024**, arXiv:2408.04755. [CrossRef]
5. Mouine, M.; Saied, M.A. Event-Driven Approach for Monitoring and Orchestration of Cloud and Edge-Enabled IoT Systems. In Proceedings of the IEEE International Conference on Cloud Computing, Barcelona, Spain, 10–16 July 2022; pp. 273–282.
6. Mekuria, D.N.; Sernani, P.; Falcionelli, N.; Dragoni, A.F. Smart Home Reasoning Systems: A Systematic Literature Review. *J. Ambient Intell. Humaniz. Comput.* **2021**, *12*, 4485–4502. [CrossRef]
7. Rahimi, H.; Trentin, I.F.; Boissier, O.; Ramparany, F. SMASH: A Semantic-enabled Multi-agent Approach for Self-adaptation of Human-centered IoT. *arXiv* **2021**, arXiv:2105.14915.
8. Papagiannidis, S.; Alamanos, E.; Marikyan, D. A Systematic Review of the Smart Home Literature: A User Perspective. *Technol. Forecast. Soc. Change* **2019**, *138*, 139–154. [CrossRef]
9. Ghai, A.S.; Rawat, V.; Gupta, V.K.; Ghai, K. Artificial Intelligence in System and Software Engineering for Auto Code Generation. In Proceedings of the 2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT), Greater Noida, India, 29–31 August 2024; pp. 1–5. [CrossRef]
10. Kanewala, T.A. Strategies and Tradeoffs in Designing and Implementing Embedded DSLs. Ph.D. Thesis, Indiana University, Bloomington, IN, USA, 2014.
11. Zhong, L.; Wang, Z. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. In Proceedings of the AAAI Conference on Artificial Intelligence, Vancouver, BC, Canada, 20–27 February 2024; Volume 38, pp. 21841–21849.
12. Sevenhuijsen, M.; Etemadi, K.; Nyberg, M. VeCoGen: Automating Generation of Formally Verified C Code with Large Language Models. *arXiv* **2024**, arXiv:2411.19275.
13. Ramachandran, A. Advancing Retrieval-Augmented Generation (RAG): Innovations, Challenges, and the Future of AI Reasoning. Available online: https://www.researchgate.net/publication/388722115_Advancing_Retrieval-Augmented_Generation_RAG_Innovations_Challenges_and_the_Future_of_AI_Reasoning (accessed on 1 April 2025).
14. Almutairi, R.; Bergami, G.; Morgan, G. Advancements and Challenges in IoT Simulators: A Comprehensive Review. *Sensors* **2024**, *24*, 1511. [CrossRef] [PubMed]
15. Nayak, S.; Patgiri, R.; Waikhom, L.; Ahmed, A. A review on edge analytics: Issues, challenges, opportunities, promises, future directions, and applications. *Digit. Commun. Netw.* **2024**, *10*, 783–804. [CrossRef]
16. Hall, R.S.; Cervantes, H. An OSGi Implementation and Experience Report. In Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC 2004), Las Vegas, NV, USA, 5–8 January 2004; pp. 394–399.
17. OSGi Modularity and Services-Tutorial. Available online: <https://www.vogella.com/tutorials/OSGi/article.html> (accessed on 21 December 2024).
18. Cheng, S.-T.; Wang, C.-H.; Horng, G.-J. OSGi-based smart home architecture for heterogeneous network. *Expert Syst. Appl.* **2012**, *39*, 12418–12429. [CrossRef]
19. Noaman, M.; Khan, M.S.; Abrar, M.F.; Ali, S.; Alvi, A.; Saleem, M.A. Challenges in Integration of Heterogeneous Internet of Things. *Sci. Program.* **2022**, *2022*, 1–14. [CrossRef]
20. Wu, X.; Duan, R.; Ni, J. Unveiling security, privacy, and ethical concerns of ChatGPT. *J. Intell. Inf.* **2024**, *2*, 102–115. [CrossRef]

21. Gama, K.; Donsez, D. A Survey on Approaches for Addressing Dependability Attributes in the OSGi Service Platform. Available online: https://www.researchgate.net/figure/Basic-Service-Oriented-Architecture-In-the-case-of-OSGi-the-binding-establishes-a_fig1_220630771 (accessed on 12 October 2024).
22. MQTT. The Standard for IoT Messaging. Available online: <https://mqtt.org/> (accessed on 1 April 2024).
23. Pinecone Docs. The Vector Database to Build Knowledgeable AI: Pinecone. Available online: <https://docs.pinecone.io/guides/get-started/overview> (accessed on 2 August 2024).
24. Pretrained Models-Sentence Transformers Documentation. Available online: https://www.sbert.net/docs/sentence_transformer/pretrained_models.html (accessed on 14 November 2024).
25. Apache Felix OSGi Tutorial: Apache Felix. Available online: <https://felix.apache.org/documentation/tutorials-examples-and-presentations/apache-felix-osgi-tutorial.html> (accessed on 15 July 2024).
26. OSGi Working Group. Getting Started with OSGi: The Eclipse Foundation. Available online: <https://www.osgi.org/resources/where-to-start/> (accessed on 9 June 2024).
27. OpenAI. API Platform | The Most Powerful Platform for Building AI Products. Available online: <https://openai.com/api> (accessed on 16 October 2024).
28. Maven Repository: Search/Browse/Explore. Available online: <https://mvnrepository.com/> (accessed on 1 June 2024).
29. Bndtools. OSGi Starter. Available online: <https://bndtools.org/workspace/osgi-starter.html> (accessed on 20 May 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.