Spec2FaaS: Using AI Agents for Autonomous Serverless Function Development and Deployment

Martina Lupini
Tor Vergata University of Rome
Italy
martina.lupini@alumni.uniroma2.eu

Gabriele Russo Russo
University of Rome Tor Vergata
Italy
russo.russo@ing.uniroma2.it

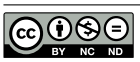
Valeria Cardellini
Tor Vergata University of Rome
Italy
cardellini@ing.uniroma2.it

Abstract

The Function-as-a-Service (FaaS) serverless paradigm is increasingly popular, allowing users to focus on application development and abstracting away operational aspects (e.g., auto-scaling), which are automatically managed by the underlying platform. The advent of Large Language Models (LLMs) capable of code generation raises the question of whether FaaS can be extended further to abstract away the coding effort from users as well. Although recent studies have investigated the use of LLMs for function generation and deployment, they do not provide an end-to-end pipeline spanning from user requirements to function deployment.

To fill this gap, this paper introduces a novel multi-agent system, Spec2FaaS, that autonomously turns natural language specifications into validated deployment on an open-source FaaS platform. Spec2FaaS decomposes development into specialized agents for code generation, test synthesis, execution, debugging, and deployment, enabling fully automated and self-correcting function development. Evaluated on the HumanEval+ benchmark, Spec2FaaS achieves competitive performance, demonstrating that fully autonomous, end-to-end serverless engineering is feasible and promising, with 90% of the functions successfully deployed and executed.

ACM Reference Format:
Martina Lupini, Gabriele Russo Russo, and Valeria Cardellini. 2026. Spec2FaaS: Using AI Agents for Autonomous Serverless Function Development and Deployment. In *The 20th ACM International Conference on Distributed and Event-based Systems (DEBS '26), June 23–26, 2026, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3809481.3812613



DEBS '26, Lisbon, Portugal

ACM ISBN 979-8-4007-2693-4/2026/06
https://doi.org/10.1145/3809481.3812613

1 Introduction

The Function-as-a-Service (FaaS) paradigm has been reshaping the way cloud-native applications are designed and deployed [14]. By decoupling application logic from infrastructure management, FaaS enables developers to focus exclusively on implementing stateless functions while delegating operational concerns – such as resource provisioning, auto-scaling, fault tolerance, and monitoring – to the underlying serverless platform. FaaS significantly reduces operational complexity and accelerates development cycles, while also enabling fine-grained pricing and seamless scalability. As a result, FaaS has gained widespread adoption across domains ranging from data processing pipelines to Internet-of-Things backends and real-time analytics [2, 6].

Despite these advances, the serverless model still presupposes that users possess the expertise required to design, implement, and optimize application logic. While infrastructure management is abstracted away, the burden of coding, debugging, and maintaining functions remains firmly in the hands of developers. This requirement can represent a non-trivial barrier, particularly for domain experts whose primary competencies lie outside software engineering. The recent emergence of Large Language Models (LLMs) with advanced code-generation capabilities introduces a compelling new dimension to this landscape. Beyond conversational applications, these models have demonstrated strong capabilities in program synthesis, code completion, refactoring, and bug fixing [8]. As a result, LLMs are increasingly used as coding assistants that translate high-level specifications into executable source code [1, 10, 21].

Early research primarily focused on improving standalone LLM-based code generation through prompt engineering, automatic correction and repair mechanisms. More recently, attention has shifted toward *AI agents* [20], software entities that leverage LLMs as reasoning cores while incorporating planning, memory, and tool invocation capabilities. By enabling interaction with external environments, agents extend LLM-based generation into action-oriented workflows. Typically, AI agents are designed to perform simple, well-defined tasks. To address more complex problems, *multi-agent* systems have emerged. These systems orchestrate multiple specialized agents that collaborate toward a shared objective,

decomposing complex processes into modular, role-driven subtasks. This paradigm has been adopted in recent work to construct automated code generation pipelines in which distinct agents emulate software development roles such as coding, testing, and debugging (e.g., [10, 18]).

Recent studies (e.g., [1, 21, 22]) have explored the use of LLMs to generate serverless functions. However, these works usually rely on a single LLM and do not use standardized datasets for evaluation. Moreover, while LLMs are used to automate code generation, function deployment is usually not handled by the LLM and left to the user. As a consequence, the transition from LLM-generated code to fully tested and autonomously deployed production-ready serverless functions remains unaddressed.

The goal of this study is to bridge this gap by proposing and evaluating a multi-agent system, called **Spec2FaaS**, that automates the full pipeline from **spec**ification to deployment on a **FaaS** platform. The system is built on a multi-agent architecture orchestrated through *AutoGen* [16]. Spec2FaaS incorporates an iterative self-debugging loop in which generated code is automatically executed against synthesized tests and refined until correctness criteria are met. Moreover, unlike prior works that restrict agentic systems to function synthesis, Spec2FaaS extends agent autonomy to the deployment phase. Deployment is handled end-to-end by a dedicated agent that automatically determines execution parameters such as CPU and memory allocation and triggers the deployment in Serverledge [19], an open-source FaaS framework, without human intervention.

The system is extensively evaluated using the HumanEval+ benchmark[1], with particular emphasis on code correctness, deployment success rate and the ability of Spec2FaaS to automatically test and fix code. This evaluation provides empirical evidence of the feasibility of fully autonomous serverless lifecycle management, while also highlighting limitations, especially as regards the generation of meaningful and accurate tests, which remains a challenging task that can negatively impact the accuracy of the whole system.

Our key contributions can be summarized as follows:

- We design a multi-agent architecture, Spec2FaaS, for the autonomous generation, testing, and deployment of serverless functions (Sec. 3);
- We implement Spec2FaaS on top of AutoGen and integrate it with Serverledge for automated function deployment (Sec. 4);
- We evaluate Spec2FaaS on HumanEval+, an established public dataset for code generation, assessing both the accuracy and the computational cost of our approach (Sec. 5).

[1]https://evalplus.github.io/

2 Related Work

Recent advances in LLMs have revolutionized automated code generation, enabling systems to produce syntactically correct and semantically meaningful source code directly from natural language specifications. The opportunities and challenges of LLM-based code generation have been the subject of several research works in the last couple of years, as recently surveyed in [11].

Many studies leveraged prompt engineering to enable iterative and self-improving code generation. Self-Debugging [4] allows an LLM to iteratively debug its own generated code by producing explanations or reasoning over sample inputs and using the resulting feedback, either self-generated or test-based, to refine the solution, achieving 73.78% pass@1 on the HumanEval benchmark. Here, pass@1 represents the proportion of problems solved correctly by the model's final refined output across a single evaluation run. SelfEvolve [12] separates code generation and correction into two phases, where example tests provided in the function description are executed and their outputs used as feedback to iteratively refine the code, achieving a higher performance of 78.05% pass@1 on HumanEval. OpenCodeInterpreter [25] adopts an iterative process where code is tested after generation and then revised based on execution results. Beyond automated testing, it also relies on synthetic human feedback to guide the correction process. CodeCoT [9] integrates Chain-of-Thought reasoning with execution-based self-correction, obtaining 79.3% pass@1 on HumanEval. This approach relies on executing the generated code to iteratively refine it, which, while effective, introduces potential security risks when functions are run in a local environment. Furthermore, the concurrent production of code and test cases may result in suboptimal code generation, LLM's attention is split between two distinct tasks, potentially diminishing its effectiveness in both. Finally, Self-Edit [24] employs a trained fault-aware editor that refines generated code based on execution-derived textual feedback.

Beyond individual models, recent approaches have explored agentic AI systems to support code generation. For example, AgentCoder [10] is a multi-agent framework comprising programmer, test designer, and test executor agents that iteratively generate, test, and refine code through execution feedback. It reaches 91.5% pass@1 with GPT-4 on HumanEval, although it also relies on code execution in a local environment. ChatDev [18] employs 7 specialized agents coordinating via multi-turn natural language dialogues and a chat-chain workflow, with a dehallucination mechanism to reduce coding errors. In contrast, the Self-Collaboration model [5] organizes analyst, coder, and tester agents in a waterfall-style pipeline with feedback loops where the tester can report issues back to the coder for correction. The agents

interact using natural language, and coordination is achieved via a shared workspace that stores intermediate artifacts such as plans and code. Notably, the code is never actually executed, but only evaluated by the tester. Other significant frameworks include MetaGPT [7], which simulates the full software development lifecycle through five role-based agents (Product Manager, Architect, Engineer, and QA Engineer). The Engineer iteratively tests, debugs, and refines the code based on the test cases designed by the QA Engineer, until all tests pass or a maximum number of iterations is reached. The framework achieves 87.7% accuracy on HumanEval. Finally, the RGD framework [13] leverages three specialized LLMs (Guide, Debug, Feedback) with execution-based refinement and memory reuse of past successful generations, obtaining 97.6% pass@1 on HumanEval. Despite these advancements, a common limitation remains: all the analyzed works rely on ad hoc solutions rather than using existing agent orchestration frameworks.

Experimental studies have recently begun to explore the potential of combining AI-driven code generation with FaaS architectures. In LLM4FaaS [21], users express desired functionality through natural language, which is translated into executable serverless functions by an LLM and deployed on a FaaS platform. The architecture consists of three main components: an LLM for code generation, a FaaS platform for execution and management, and an intermediate bridge that orchestrates the interaction between the two. The LLM-FaaS bridge is equipped with a *prompt constructor* for the LLM and a *function deployer* for the function deployment. Instead of relying on an established benchmark, the evaluation is based on a custom dataset obtained through a questionnaire, in which the participants described how a smart home system should perform a set of automation tasks of increasing difficulty. Generated functions are evaluated in terms of syntactic correctness, defined as successful execution without runtime errors, and semantic correctness, measured by passing predefined test cases. The study does not incorporate iterative refinement or debugging feedback. Another study [1] examines the ability of LLMs to generate serverless functions comparable to human-written code. By regenerating functions from open-source projects using various prompt configurations (e.g., README-based zero-shot vs. architectural few-shot), the study found that models like DeepSeek-Coder-V2 and GPT-4 achieve an 80% pass rate. However, unlike the work presented in this paper, this approach relies on automatically constructed technical prompts and lacks automated deployment or self-improvement mechanisms. Interestingly, ReFaaS [22] shifts the focus to *energy debt* by using LLMs to translate serverless functions into more energy-efficient programming languages. Its translation pipeline comprises generation, build, and testing phases, each equipped with verification and recovery mechanisms to address compilation and logical errors. While it verifies behavioral equivalence between implementations (e.g., Python to Go), it requires user-provided test inputs rather than generating them automatically.

Table 1: Comparison of relevant related works.

	Correction mechanism	Code execution	Docker code execution	Test generation	Human intervention needed	Deployment (FaaS)	Multiple LLM	Multi agent
Self-Debugging [4]	✓	✓	?	✗	✓	✗	✗	✗
SelfEvolve [12]	✓	✓	?	✗	✓	✗	✗	✗
OpenCodeInterpreter [25]	✓	✓	?	✗	✓	✗	✗	✗
CodeCoT [9]	✓	✓	?	✓	✗	✗	✗	✗
SelfEdit [24]	✓	✓	✗	✗	✓	✗	✗	✗
AgentCoder [10]	✓	✓	✗	✓	✗	✗	✓	✓
ChatDev [18]	✓	?	?	?	✗	✗	✓	✓
Self-Collaboration [5]	✓	✗	✗	✗	✗	✗	✓	✓
MetaGPT [7]	✓	✓	?	✓	✗	✗	✓	✓
RGD [13]	✓	✓	?	✗	✗	✗	✓	✓
LLM4FaaS [21]	✗	✗	✗	✗	✗	✓	✗	✗
Automatic Serverless Generation [1]	✗	✗	✗	✗	✗	✗	✓	✗
ReFaaS [22]	✓	✓	✗	✗	✓	✓	✗	✗
Spec2FaaS	✓	✓	✓	✓	✗	✓	✓	✓

Note. ? indicates that the feature is *not explicitly stated in the paper.*

Table 1 provides a summary of relevant related works and compares them with our proposed Spec2FaaS system.

3 System Design

In this section, we present the architecture of Spec2FaaS. It is made up of seven specialized agents, each responsible for a specific task and interacting with each other to complete the function generation and deployment workflow. The system is designed with a hierarchical structure, where supervisor agents orchestrate and control the activation of subsequent steps in the workflow. The agents in Spec2FaaS interact through well-defined, synchronous, point-to-point messages. Nevertheless, the system remains flexible and scalable, as agents can be added or removed with minimal impact on its overall behavior, requiring only limited adjustments to the message-passing logic. Despite synchronous communication, the workflow does not follow a fully deterministic execution path: while the ordering of steps is fixed, the actual sequence of executed steps depends on the outputs generated by the LLMs and on the outcomes of code execution and tool invocation, as explained in the following.

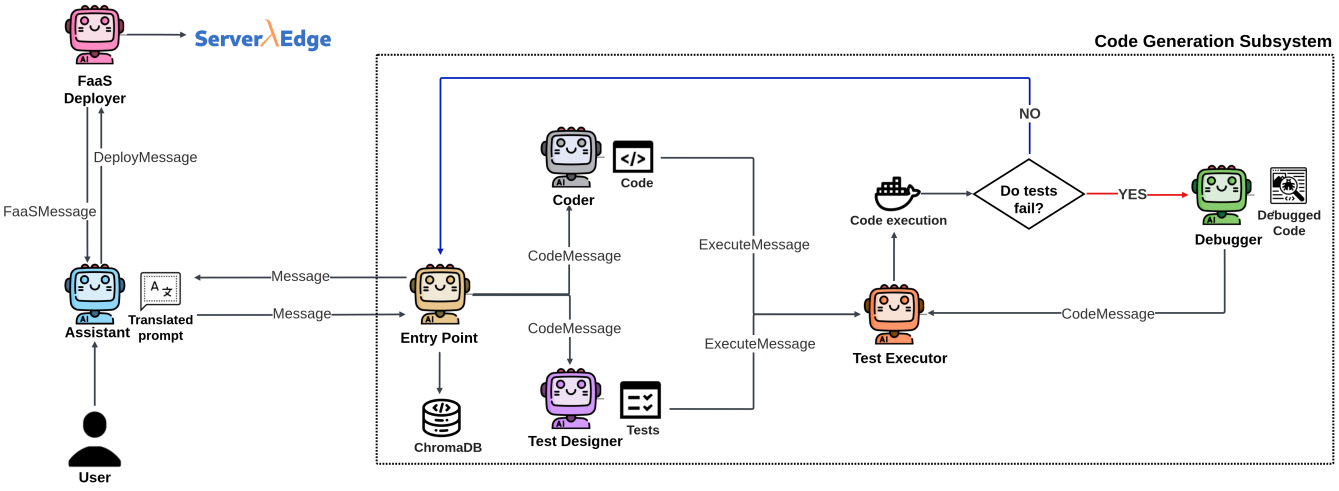


Figure 1: Architecture of Spec2FaaS.

Figure 1 illustrates the overall architecture of Spec2FaaS. Each agent in Spec2FaaS is built around two essential elements: (i) a *LLM* acting as the "brain" of the agent, and (ii) one or more *message handlers*, which are dedicated methods that handle a particular type of message and enable interaction among agents. Each agent is associated with its own LLM instance, independent of those used by other agents. The LLM leverages a *system prompt* based on role prompting to assign the agent's specific role and tasks (e.g., generating code or debugging).

Depending on its assigned role, each agent may also be equipped with additional capabilities, such as:

- **Tools**: agents can exploit *tool calling* to interact with external systems/tools (e.g., FaaS Deployer Agent uses tools to deploy generated functions);
- **Code Execution Capabilities**: agents, such as the Test Executor, possess execution capabilities (e.g., Docker-based) to run generated code;
- **Retrieval Augmented Generation** (RAG): to handle high-level specifications and domain-specific knowledge, for example, the Entry Point Agent utilizes RAG to query local documentation or architectural patterns.

By decoupling the LLM instance from the agent toolset, Spec2FaaS achieves a highly modular design. In the following, we describe the role and capabilities of each agent in detail.

3.1 Assistant Agent

The Assistant Agent is the entry point of the system. It receives the prompt from the user and analyzes it in order to decide what is the next step to take. Three outcomes are possible:

- The prompt is *unclear*: in this case, the Assistant asks the user for more information or clarification.
- The prompt contains the *source code of a function* already implemented: in this case, the Assistant directly forwards the code to the FaaS Deployer Agent for deployment.
- The prompt contains a natural-language *specification* of a function. If the specification is provided in a language other than English, the Assistant translates it before passing the task in a message to the Entry Point Agent to initiate the coding process. The translation of the specification into English is motivated by the fact that LLMs are mainly trained on English-language corpora. Therefore, by standardizing the input in English at the start of the pipeline, Spec2FaaS increases the likelihood of generating higher-quality code.

3.2 Entry Point Agent

The Entry Point Agent is the first agent in the coding workflow and the only coding agent that interacts with agents outside of the code–generation subsystem. In particular, it communicates with the Assistant Agent by receiving the translated specification and returning the generated function once completed.

The Entry Point Agent performs four main tasks:

(1) leveraging RAG, it searches for relevant pseudocode snippets that can be useful to implement the requested function;
(2) it defines the signature of the function to be implemented. This step is crucial, as the same signature must be used consistently by the Coder Agent and the Test Designer Agent to ensure coherence between the generated code and the corresponding test suite;

(3) it forwards a message to the Coder Agent and the Test Designer Agent, containing the selected function signature together with any retrieved pseudocode snippets, that will later be included as memory context in their prompts;
(4) after the code generation process is complete, the Entry Point Agent returns the final outcome to the Assistant Agent (i.e., either the generated function or an error message).

3.3 Coder Agent

The Coder Agent is responsible for the fundamental task of generating the code of the function given its specification, signature, and additional contextual information retrieved from memory. The memory context retrieved during the Entry Point execution phase is injected into the prompt prior to invoking the LLM, allowing the agent to decide whether and to what extent such context should be incorporated into the generated code. Once the function has been generated, the Coder Agent forwards the resulting code to the Test Executor Agent by issuing a new message containing the implementation.

3.4 Test Designer Agent

The Test Designer Agent is responsible for constructing a test suite for the target function. Based on the function specification, its signature, and the retrieved memory context, the agent generates a set of black-box tests, each expressed in the following form:

```
assert function_name(input) == expected_output, 'Test Case
                <number>: Description'
```

where the description consists of a brief explanation of the test case. Note that we do not provide the agent with the implementation of the function under test. This ensures that the Test Designer Agent focuses exclusively on the expected external behavior, rather than on (possibly misleading) implementation details. This design choice promotes specification-driven testing with the goal of increasing the robustness of the generated test cases.

Once the test suite has been generated, the Test Designer Agent sends a new message to the Test Executor Agent, containing the generated tests.

3.5 Test Executor Agent

The Test Executor Agent checks the correctness of the function generated by the Coder Agent by running it against the test suite produced by the Test Designer Agent. After the code execution, the Test Executor Agent analyzes the resulting output. If any `Error` is detected, the function is considered incorrect; otherwise, it is correct and the final implementation of the function goes back to the Entry Point Agent. When an error occurs, the Test Executor Agent sends a message to the Debugger Agent, including the function specification, the current implementation, and the reported error. It then waits for a response containing the revised code. Once the debugged version of the code is received, the Test Executor Agent runs it again against the same tests. If the tests fail, the debugging process is repeated. This loop continues until the function passes all tests or the maximum number of attempts (i.e., 10 in our study) is reached.

3.6 Debugger Agent

As the name suggests, the Debugger Agent's task is fixing the code that fails the test suite. Based on the function specification, the function implementation, and the reported error, the Debugger Agent analyzes the failure and produces a corrected version of the code. Once the debugging step is completed, the updated implementation is returned to the Test Executor Agent.

For each function, the Debugger Agent keeps a record of previous debugging iterations, including which tests failed and the corresponding fixes that were applied. This information is used in subsequent iterations to guide the debugging process and avoid repeating the same errors.

3.7 FaaS Deployer Agent

The FaaS Deployer Agent plays a central role in the system, as it is responsible for deploying the generated function onto the target FaaS platform. As already mentioned, we chose to use Serverledge as the target framework in this study, but the design of Spec2FaaS is general enough to support alternative frameworks as well.

To perform the deployment task, the FaaS Deployer Agent relies on the `create_json_serverledge` tool that we developed. This tool is implemented as a Python function that essentially wraps a HTTP API call towards Serverledge endpoint to register a new function. The tool takes the following input parameters, which are then forwared to Serverledge:

- `code`: source code of the function to be deployed;
- `name`: name of the function (it has to be globally unique in Serverledge);
- `runtime`: function container runtime (i.e. Python 3.10);
- `memoryMB`: amount of memory, in MB, reserved to each function instance;
- `CPUDemand`: number of CPU cores (or fractions of) allocated to each function instance.

The arguments passed to the tool are dynamically selected by the FaaS Deployer Agent at invocation time, allowing the deployment configuration to be tailored to the type and behavior of the function being deployed.

The FaaS Deployer Agent receives messages from the Assistant Agent that exclusively contain the source code of the

function to be deployed. Upon arrival of a message, the agent first adds a *handler* to the source code that wraps the actual function and adheres to the function signature expected by Serverledge. Then, the agent calls the deployment tool and the output is incorporated into the LLM prompt and fed back to the model.

Based on the deployment outcome, the FaaS Deployer Agent determines the next step in the control flow: it may either terminate the deployment process upon a successful deployment, or re-invoke the tool to resolve issues, such as naming conflicts or resource allocation errors. Once its task is completed, the FaaS Deployer Agent returns a message to the Assistant Agent, containing the outcome of the deployment process, the names of the function arguments, or an error message if the deployment process did not complete successfully.

4 Implementation

We implemented Spec2FaaS[2] relying on AutoGen [16, 17] for orchestration. AutoGen is a multi-agent framework based on event-driven communication, where agents interact exclusively through structured message exchange. Message routing and delivery are managed by a runtime layer without imposing a centralized control logic.

Among the several orchestration frameworks available (e.g., LangGraph, CrewAI), we selected AutoGen because it enables emergent orchestration: the coordination of tasks arises dynamically from inter-agent communication rather than from a predefined procedural workflow. The system behavior is therefore shaped by the conversational exchanges among agents, making the coordination process more flexible and conceptually closer to human collaborative problem-solving.

Moreover, AutoGen promotes high modularity: each agent is implemented as an autonomous unit, with its own logic and message handlers. This design makes the system easily extensible or reducible, as agents can be added, modified, or removed without affecting the overall orchestration mechanism. Such architectural decoupling was essential to support the iterative refinement loops required for code generation and to enable future extensions of the Spec2FaaS architecture with additional specialized roles.

4.1 Integration with Serverledge

The target FaaS platform for this paper is Serverledge [19]. Unlike traditional cloud-centric FaaS platforms (e.g., Apache OpenWhisk, OpenFaaS), Serverledge is designed for the edge-cloud continuum and adopts a fully decentralized architecture. This design allows function invocation requests to be served at the edge or dynamically offloaded, either horizontally to neighboring edge nodes or vertically to cloud nodes, based on real-time resource availability and latency requirements.

It is worth noting that while Serverledge was selected for its native edge-cloud orientation, the modular architecture of *Spec2FaaS* ensures that the underlying FaaS platform can be seamlessly replaced with alternative FaaS frameworks by simply updating the FaaS Deployer Agent.

The integration of Spec2FaaS and Serverledge was straightforward, as we only had to implement the aforementioned `create_json_serverledge` tool for the FaaS Deployer Agent. The tool is just a Python script wrapping an HTTP API call to Serverledge.

5 Experimental Evaluation

We perform a set of experiments to evaluate Spec2FaaS and the agent it comprises. Specifically, we aim to answer the following key questions:

- **Q1** To what extent is the Coder Agent capable of generating functionally correct functions?
- **Q2** Are the tests generated by the Test Designer Agent accurate?
- **Q3** Is the self-improving mechanism effective in correcting generated code?
- **Q4** Does the FaaS Deployer Agent reliably deploy functionally correct code to the target serverless environment?
- **Q5** Can Spec2FaaS successfully deploy and correctly execute the generated functions?
- **Q6** What proportion of deployed and executed functions are actually correct?
- **Q7** How does the system perform in terms of token usage and computational costs?

5.1 Experimental Setup

5.1.1 Dataset. For evaluation, we used the HumanEval+ [15] dataset, which has become the de facto benchmark for assessing LLMs coding abilities. The dataset consists of 164 human-written Python functions. Each entry includes a natural language specification, a function signature, a reference human-written implementation, and an extensive test suite (on average 764 tests per function), allowing for a rigorous validation of the code.

5.1.2 Models. We considered different LLM models for comparison. The selection of the LLMs was based on two main criteria: whether the model is open-weight or proprietary, and whether it is a general-purpose model or specifically designed for code-related tasks. The LLMs we selected are reported in Table 2. Among them, Gemini models were used through the APIs provided by Google, whereas the other

[2] https://github.com/martinalupini/Spec2Faas

models, alongside the other Spec2FaaS components, were deployed on a local server equipped with a NVIDIA RTX 4000 GPU.

5.1.3 Experimental Workflow. Except for the evaluation of the complete system, agents are tested in isolation to observe their behavior without the influence of other components. This choice is motivated by the nature of multi-agent systems, where agents are strongly interconnected and failures are often difficult to trace back to a single source.

Moreover, for the experimental evaluation, RAG is not enabled. Although it would have been possible to inspect the functions contained in the dataset and manually construct auxiliary pseudocode to be used as retrievable context, this approach was deliberately avoided. Injecting handcrafted pseudocode could have biased the evaluation by partially encoding solution-specific information, thereby reducing the ability to assess the agents' standalone reasoning and code generation capabilities. For this reason, the experiments are conducted without retrieval, focusing on the agents' behavior when operating solely on the input specification.

5.2 Results

5.2.1 Q1: To what extent is the Coder Agent capable of generating functionally correct functions? We first evaluate the ability of the Coder Agent to generate functions that are correct from both a syntactic and semantic perspective, i.e., the generated function runs without errors and behaves as required by the specification. In this experiment, performance is evaluated using pass@1 [3], a standard metric commonly adopted in the code generation literature, which denotes the probability of the model output being correct after a single generation attempt.

The results, illustrated in Figure 2, show that Gemini Pro 2.5 achieves the highest pass@1 score (0.96), corresponding to 158 correctly generated functions. It is followed by Qwen 2.5 Coder (32B), which obtains a pass@1 of 0.87, meaning that 142 functions are implemented correctly. No other model matches the performance of Gemini Pro 2.5. However, several open-weight LLMs, such as Qwen 2.5 Coder (32B) and Qwen 3, outperform Gemini Flash 2.0, whose results are comparable to those of Qwen 2.5 Coder (7B). Overall, within the scope of this experiment, open-weight LLMs prove to be competitive with proprietary models that do not represent the current state of the art.

Because of the non-determinism of LLMs in replying to identical requests, we also verified the impact of such variability on the observed results. To do so, we performed independent runs of the same model under identical experimental settings. Specifically, the experiment presented above was repeated 20 times using Deepseek Coder V2 as the Coder Agent, which was selected due to its low inference latency.

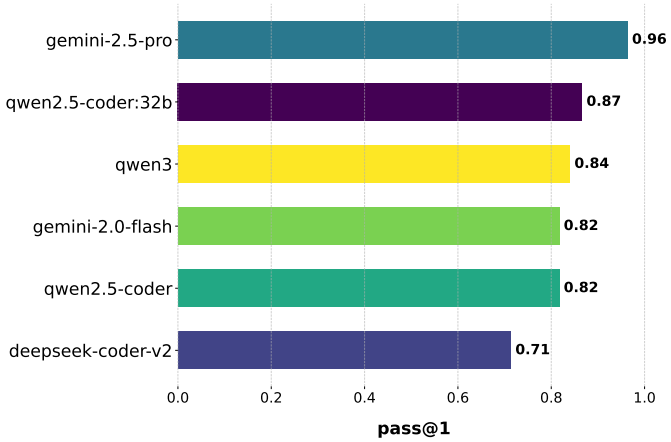


Figure 2: Comparison of different LLMs as the reasoning component of the Coder Agent.

As shown in Table 3, all the metrics of interest (i.e., generated tokens and inference time in addition to pass@1) demonstrate remarkably low variability. Given the observed stability across repeated executions, the differences reported between models in the experiments are unlikely to be driven by random fluctuations, but rather reflect systematic differences in model capabilities and behavior under the proposed experimental setup. For this reason, given the significant costs and inference times that replicating all the experiments would require, we will consider single inference attempts in the rest of this section.

5.2.2 Q2: Are the tests generated by the Test Designer Agent accurate? This question examines the ability of the Test Designer Agent to produce accurate tests based on a function specification. A test is considered inaccurate if it violates the specification, is malformed, or does not correctly reflect the intended behavior of the function. Test accuracy is evaluated by executing the generated tests against the canonical solution provided in the dataset, which is assumed correct.

We again consider pass@1 as the key metric in this experiment. As reported in Fig. 3, all the LLMs we initially selected (see Table 2) exhibit lower accuracy than expected, with the best performance achieved by Qwen3 with a pass@1 value of 0.74, and several models not reaching 0.5. To further analyze these results, we compared them to those reported in AgentCoder [10] (summarized in Table 4). We note that our observations are in line with their results. In fact, the Qwen3 model used in this experiment is the 8B-parameter version, which exhibits lower performance compared to the largest 235B-parameter variant. The latter has been shown to achieve performance comparable to, or even exceeding, that of powerful proprietary LLMs, as reported in the official Qwen3 technical report [23].

Table 2: Overview of the selected LLMs.

Model	Company	Parameters	Size	Context	Input
Deepseek Coder V2	Deepseek	16B	>8.9GB	160K	Text
Gemini 2.0 Flash	Google	-	-	1M	Multimodal
Gemini 2.5 Pro	Google	-	-	1M	Multimodal
Qwen 3	Alibaba Cloud	8B	5.2GB	40K	Text
Qwen 2.5 Coder 32:b	Alibaba Cloud	32B	20GB	32K	Text
Qwen 2.5 Coder	Alibaba Cloud	7B	4.7GB	32K	Text

Table 3: Standard deviation and coefficient of variation (CV) of key evaluation metrics across 20 independent runs of the experiment related to Q1 using Deepseek Coder V2 as the Coder Agent.

Metric	Std. Dev.	CV
Pass@1	0.015	2.14
Tokens	2.509	0.61
Inference Time (s)	0.031	2.46

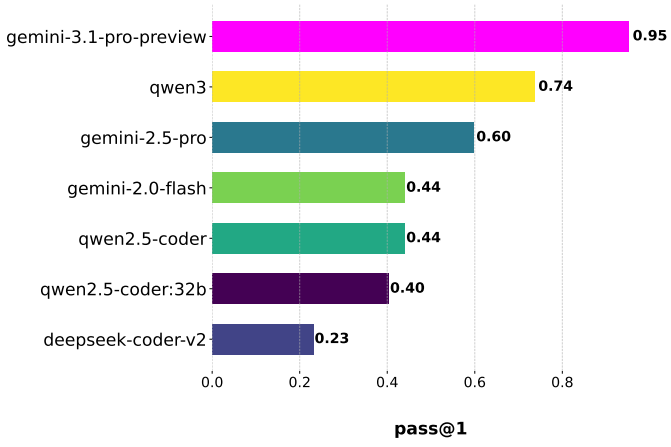


Figure 3: Comparison of different LLMs as the reasoning component of the Test Designer Agent, evaluated using the pass@1 metric.

To validate our hypothesis that more recent and capable LLMs could achieve better performance, we repeated the evaluation of the Test Designer Agent considering the preview version of Gemini 3.1 Pro. Figure 3 confirms that in this case the agent reaches a 0.95 pass@1 value, largely outperforming the other models. Due to budget limitations, the use of this model has been limited to this experiment.

In contrast, Spec2FaaS achieves higher line coverage compared to other approaches reported in the literature. This suggests that our prompt engineering techniques were effective in generating comprehensive test suites, even when employing LLMs that do not represent the current state of the art. These results underscore the importance of the architectural orchestration over mere model scale in achieving robust code validation.

Table 4: Comparison of different frameworks in terms of test generation accuracy and line coverage on HumanEval, as reported in the Agent Coder paper [10].

Models	Pass@1	Line coverage
GPT-3.5-turbo	47.0	70.2
CodeCoT	67.1	77.2
AgentCoder (GPT-3.5-turbo)	87.8	85.7
MetaGPT (GPT-4)	79.3	81.7
AgentCoder (GPT-4)	89.6	91.7
Spec2FaaS (Qwen3 8B)	**74.0**	**95.8**

5.2.3 Q3: Is the self-improving mechanism effective in correcting generated code? This question evaluates whether the self-improving mechanism is able to effectively identify and correct errors in the generated code.

To evaluate this capability, the analysis focuses on three representative Coder Agents chosen from the results of RQ1 (see Figure 2): Gemini Pro 2.5, which emerged as the best-performing code generator; DeepSeek-Coder v2, which showed the weakest performance; and Gemini Flash 2.0, which achieved intermediate results. For each of these coders, all the remaining models are alternatively employed as the Debugger Agent. This design allows assessing whether debugging performance depends on the specific coder–debugger pairing, or whether there exists a model that consistently performs best as a debugger, independently of the coder.

The results are reported in terms of *debugging gain*. Debugging gain is defined as the difference between the number of functions that pass the test suite after the self-improving mechanism and the number of functions that already passed the tests immediately after generation by the coder, without any debugging step:

$$\text{Debugging Gain} = \#\text{Passed}_{\text{after}} - \#\text{Passed}_{\text{before}}$$

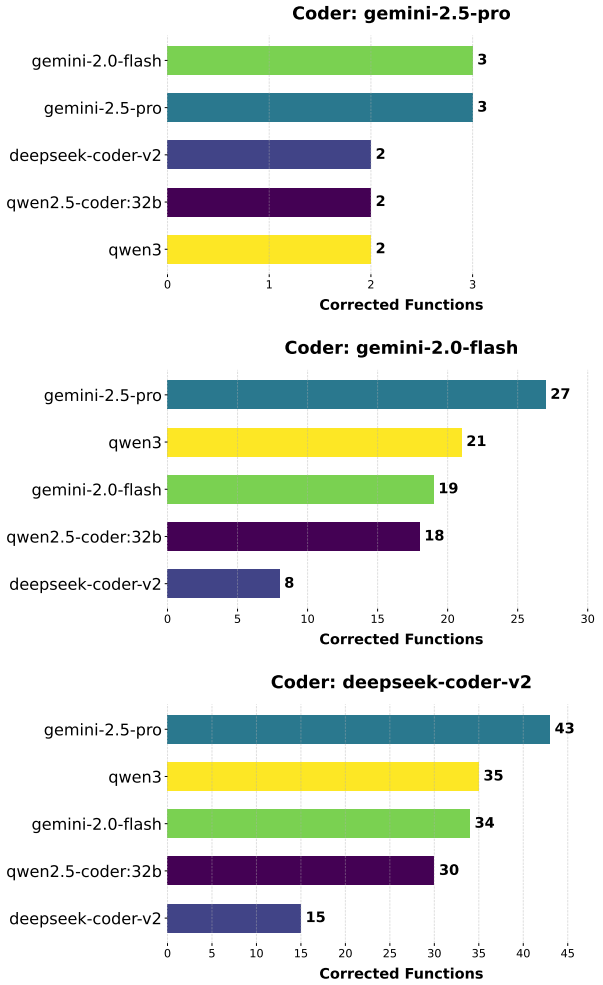


Figure 4: Comparison of different LLMs as the reasoning component of the Debugger Agent, considering three different models for the Coder Agent.

As shown in Figure 4, Gemini Pro 2.5 achieves the highest performance when acting as the Debugger Agent, followed by Qwen 3 and Gemini Flash 2.0. Overall, the self-improving mechanism proves to be effective, as it increases the number of correct functions by fixing a relevant fraction of the initially failing ones. An exception emerges when Gemini Pro 2.5 is used as the coder: in this case, several models exhibit difficulties in correcting its outputs. A possible interpretation is that models may struggle to improve or repair code generated by an already powerful LLM.

5.2.4 Q4: Does the FaaS Deployer Agent reliably deploy functionally correct code to the target serverless environment? This question evaluates the performance of the FaaS Deployer Agent, focusing on its ability to successfully deploy correct functions. We evaluate the FaaS Deployer Agent using all the considered models, with the exception of Qwen 2.5 Coder (32B), which was unable to deploy any function. Figure 5 reports the results in terms of number of deployed functions and successfully executed functions. These results demonstrate that the FaaS Deployer Agent is capable of successfully deploying functions, with the best-performing model, Gemini Pro 2.5, achieving a 100% succcess rate. Moreover, the number of functions that are correctly executed after deployment is also very high (e.g., 91% for Gemini Pro 2.5). This provides further evidence that the FaaS Deployer Agent does not simply invoke the deployment tool, but in most the cases it also correctly constructs the function handler and appropriately selects the resource parameters to be assigned to the function, such as CPU and memory.

There are some cases in which the deployment succeeds, but the subsequent execution of the function results in an error. A closer inspection of these failures reveals that they are mainly caused by malformed handlers (for instance, using incorrect parameter names) or by suboptimal resource allocation, either insufficient resources, or excessively high values that overload the system. For future work, we plan to integrate additional mechanisms to automatically identify and fix such minor issues.

5.2.5 Q5: Can Spec2FaaS successfully deploy and correctly execute the generated functions? This question evaluates the end-to-end performance of the framework. Given a function specification, it measures whether the system is able to generate, test, debug, and deploy a function that executes correctly in the serverless environment. The number of functions that are successfully deployed and correctly executed is adopted as the final evaluation criteria of the entire system, as it directly represents the effective outcome of the pipeline and the actual service delivered by Spec2FaaS. Regardless of how many functions are generated, tested, or debugged in intermediate stages, what ultimately matters in a real scenario is whether a function can be deployed and run correctly. From this perspective, the number of deployed and executed functions captures the real operational value of the system and is therefore the most meaningful indicator of its overall performance.

To answer this research question, two configurations have been considered, as reported in Table 5. Configuration A combines the best-performing model for each individual task within the pipeline[3] Conversely, configuration B is not optimal and including this configuration allows to further validate the results of the Spec2FaaS system, showing that the observed outcomes do not depend only on the use of the best LLMs.

[3]Except for the Test Designer Agent, where the best results in Fig. 3 were achieved by Gemini 3.1 Pro. The use of that model was limited to that experiment due to budget limitations.

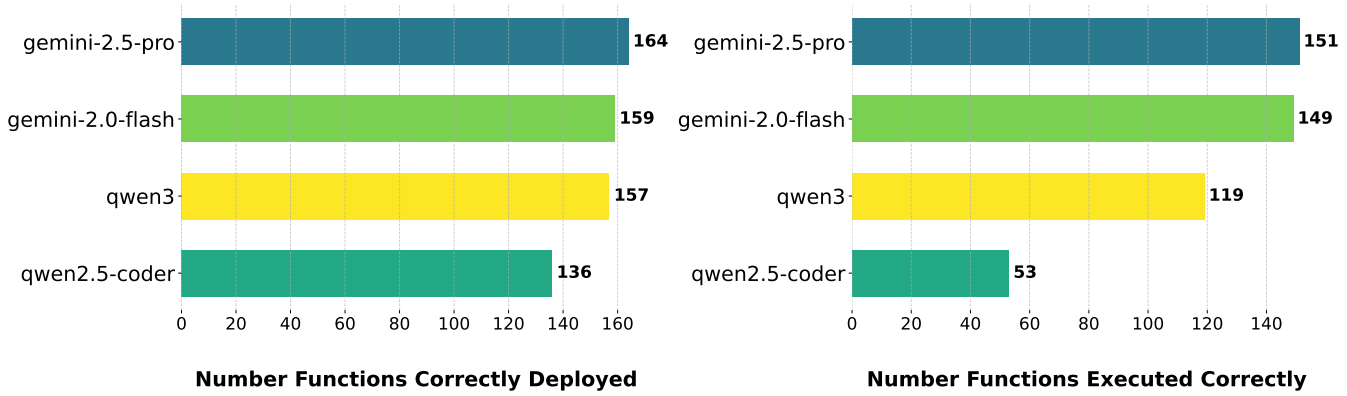


Figure 5: Comparison of different LLMs as the reasoning component of the Faas Deployer Agent in terms of number of function correctly deployed and executed.

Table 5: Model configurations used in the end-to-end experiments.

Agent	Configuration A	Configuration B
Assistant	Gemini 2.5 Pro	Gemini 2.0 Flash
Coder	Gemini 2.5 Pro	DeepSeek Coder v2
Debugger	Gemini 2.5 Pro	Gemini 2.5 Pro
Entry Point	Gemini 2.0 Flash	Gemini 2.0 Flash
FaaS Deployer	Gemini 2.5 Pro	Gemini 2.0 Flash
Test Designer	Qwen 3	Qwen 3
Test Executor	Gemini 2.0 Flash	Gemini 2.0 Flash

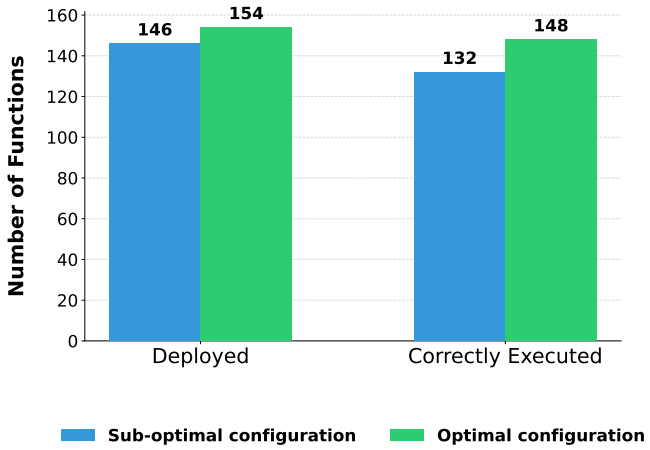


Figure 6: Comparison of different system configurations (optimal and sub-optimal) in terms of number of functions deployed and correctly executed. Optimal refers to the system configuration that uses the best-performing model for each task. Sub-optimal refers to a configuration that does not use the best-performing models for all tasks.

As shown in Figure 6, the number of functions that are finally deployed and correctly executed is quite high. In the optimal configuration, 154 out of 164 functions are successfully deployed, corresponding to nearly 94% of the total. Among these, 148 functions are executed correctly. The results obtained with the sub-optimal configuration are only slightly worse. As expected, using models that are not top performers for all tasks leads to a modest reduction in performance. However, the fact that the difference between the two configurations is not substantial indicates that the Spec2FaaS architecture is sufficiently robust to maintain high performance even when different LLMs are used.

5.2.6 Q6: What proportion of deployed and executed functions are actually correct? This question focuses on the correctness of the deployed functions. Since the generated tests may be not sufficiently comprehensive or, in some cases, even incorrect, it is not guaranteed that the functions entering the debugging phase were originally incorrect. Conversely, incorrect implementations may still pass the generated tests and, therefore, be deployed on the FaaS platform. The results reported in Figure 7, presented as a Sankey diagram, confirm these concerns. From the diagram, we can observe that several originally correct implementations pass through the debugging phase and, in some cases, are even transformed into incorrect ones. At the same time, the debugging process often fails to detect incorrect implementations. Overall, the debugging process never succeeds in correcting originally wrong implementations. Only 13 incorrect implementations are identified as such, but the self-improving mechanism fails to fix them, likely because the generated tests are trivial or not sufficiently informative. In addition, 55 incorrect

implementations pass the tests undetected, while 18 correct implementations are mistakenly classified as incorrect and are therefore sent to the debugging phase.

These results further highlight the critical role of test generation, which emerged as the most difficult task for our agents and is fundamental for the success of the self-correction mechanism. In Sec. 5.2.2, we showed that exploiting the more recent Gemini 3.1 Pro model the accuracy of the generated tests dramatically improves and, thus, similar improvements can be expected in the end-to-end behavior of Spec2FaaS leveraging newer LLM releases. Besides this, alternative strategies could be considered, including human validation of the generated tests.

5.2.7 Q7: How does the system perform in terms of token usage and computational costs? The monetary cost is estimated by multiplying the token usage of the proprietary models (Gemini Pro 2.5 and Gemini Flash 2.0) by their respective per-token prices (at the time of writing). The results show that the system has an average cost of approximately 3.2$ per run in its optimal configuration. This cost is highly concentrated in the Debugger Agent, which alone accounts for the vast majority of token usage and monetary expenditure due to long contexts and iterative correction cycles.

From an energy consumption perspective, the local execution footprint is modest: the locally deployed Test Designer Agent consumes on average only 0.00274 kWh per run, even under a conservative upper-bound assumption of constant maximum GPU power draw. We cannot measure energy consumption of cloud-based models.

In terms of latency, the full workflow completes in about 6 minutes on average (384 s). The main time bottleneck lies in the execution and debugging phases, while deployment is comparatively fast, as shown in Fig. 8.

Table 6 provides additional insight on the amount of data exchanged among the agents. Overall, the system requires an average of 9.4 messages to complete a generation task, with slightly less than 7 KB of data exchanged. In the best case, when the code is correctly generated at the first attempt, communication reduces to 5.6 KB on average. In contrast, unsuccessful generations exhibit the highest communication overhead due to repeated debugging and testing rounds, reaching 17 exchanged messages on average (i.e., 20 KB of data). Overall, these results indicate that the proposed multi-agent workflow introduces modest communication overheads, with the debugging phase having again the most evident impact.

Table 6: Amount of exchanged data among agents (averages)

Case	# Messages	Data (KB)
All outcomes	9.4	6.8
Successful generation	9.0	5.6
Successful gen. with debugging	10.8	8.9
Unsuccessful generation	17.0	20.4

5.3 Discussion

The experiments conducted in isolation on each agent show that their performance is comparable to that reported in previous studies, even though those works often rely on more powerful LLMs than the ones used here. This result highlights the effectiveness of the prompt-engineering strategies we adopted and suggests that agent performance could further improve when leveraging state-of-the-art LLMs.

The experimental evaluation on the whole system shows that 89% of the specifications are translated into functions that can be correctly invoked on the FaaS platform. This result shows that Spec2FaaS is able to automate the software development lifecycle in a truly end-to-end manner and represents a promising research prototype. However, the system still presents several limitations, which are discussed below together with possible solutions.

The most evident limitation concerns the low proportion of deployed and executed functions that are actually correct. As already mentioned, this issue is mainly due to incorrect or incomplete generated test suites, which may fail to detect faulty implementations or, conversely, incorrectly flag correct ones as erroneous. However, test generation is known to achieve relatively low accuracy in the existing literature and represents a common limitation rather than an issue specific to this work. As shown in Sec. 3, experiments with the more recent Gemini 3.1 Pro model show that relying on newer LLMs can significantly improve the resulting accuracy.

Another limitation is that the system has been evaluated only on the HumanEval+ dataset. Using multiple datasets would allow the performance of Spec2FaaS to be assessed from different perspectives and consider more heterogeneous functions. Another aspect to acknowledge is that possible (direct or indirect) data contamination between the evaluated LLMs and the HumanEval+ dataset cannot be completely excluded. Although this issue lies outside the control of the proposed system, it further motivates the extension of the experimental evaluation to multiple independent datasets to strengthen the generality of the results.

Overall, the system demonstrates a reasonable trade-off between autonomy, reliability, and operational cost, although further optimizations of the test design and debugging phases would significantly improve real-world usability.

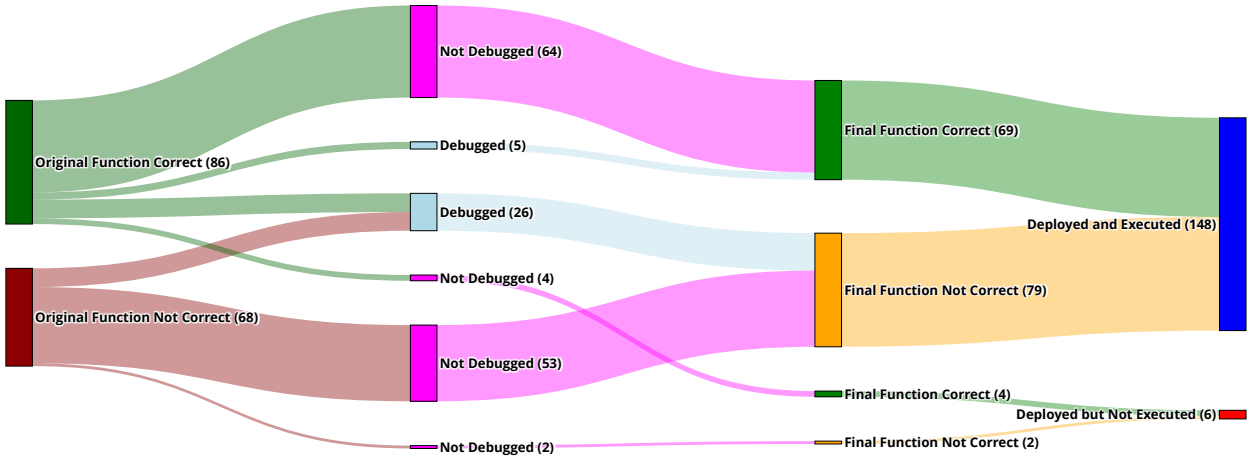


Figure 7: Sankey diagram showing the distribution of correct functions across the different stages of the Spec2FaaS pipeline.

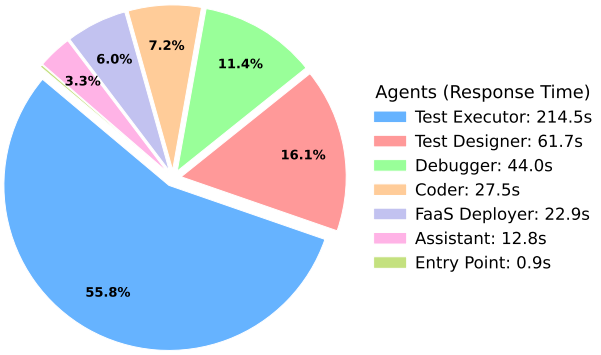


Figure 8: Breakdown of the average execution time by agent.

6 Conclusion

In this work, we proposed Spec2FaaS, a multi-agent system for the automatic generation, testing, debugging, and deployment of serverless functions. Differently from previous works, which focused either on deployment or on generating a correct function through self-correction techniques, we implement a fully automated lifecycle. Starting from a natural-language specification, the system proceeds through validation and correction phases and ultimately produces a function that is registered and ready for execution on a FaaS platform. Evaluated on the HumanEval+ dataset, Spec2FaaS demonstrates good accuracy and represents a promising research prototype for further experimentation.

Based on the limitations discussed above, we identify several directions for future work. As already mentioned, we plan to investigate strategies to mitigate the negative impact that incorrectly designed tests can have on the overall performance of Spec2FaaS. While considering newer and more powerful LLM models is likely to reduce errors without modifications to the system itself, considering human-in-the-loop scenarios might be an interesting extension. We also plan to extend the function generation ability of Spec2FaaS to *serverless workflows*, allowing for the autonomous generation of more complex applications. Furthermore, inspired by recent work like [22], we will consider integrating non-functional requirements (e.g., energy footprint) in the generation process, allowing Spec2FaaS to produce multiple implementations of the same function with different performance–energy trade-offs.

References

[1] Shrikara Arun, Meghana Tedla, and Karthik Vaidhyanathan. 2025. LLMs for generation of architectural components: An exploratory empirical study in the serverless world. In *Proc. of 2025 IEEE 22nd International Conference on Software Architecture, ICSA '25*. IEEE, 25–36. doi:10.1109/ICSA65012.2025.00013

[2] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. 2022. Serverless computing for Internet of Things: A systematic literature review. *Future Gener. Comput. Syst.* 128 (2022), 299–316. doi:10.1016/J.FUTURE.2021.10.020

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023). https://arxiv.org/abs/2304.05128

[5] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 189 (2024), 38 pages. doi:10.1145/3672459

[6] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Trans. Software Eng.* 48, 10 (2022), 4152–4166. doi:10.1109/TSE.2021.3113940

[7] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proc. of 2024 International Conference on Learning Representations, ICLR '24*. https://proceedings.iclr.cc/paper_files/paper/2024/file/6507b115562bb0a305f1958ccc87355a-Paper-Conference.pdf

[8] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (2024), 220:1–220:79. doi:10.1145/3695988

[9] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. CodeCoT: Tackling code syntax errors in CoT reasoning for code generation. *arXiv preprint arXiv:2308.08784* (2023). doi:10.48550/arXiv.2308.08784

[10] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-agent code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023). doi:10.48550/arXiv.2312.13010

[11] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.* 35, 2, Article 58 (2026), 72 pages. doi:10.1145/3747588

[12] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907* (2023). doi:10.48550/arXiv.2306.02907

[13] Haolin Jin, Zechao Sun, and Huaming Chen. 2024. RGD: Multi-LLM based agent debugger via refinement and generation guidance. In *Proc. of 2024 IEEE International Conference on Agents, ICA '24*. IEEE, 136–141.

[14] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian T. Foster, Prashant J. Shenoy, Omer F. Rana, and Andrew A. Chien. 2023. Serverless computing: What it is, and what it is not? *Commun. ACM* 66, 9 (2023), 80–92. doi:10.1145/3587249

[15] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Proc. of 37th International Conference on Neural Information Processing Systems, NIPS '23*. Curran Associates Inc., Article 943, 15 pages.

[16] Microsoft. 2023. AutoGen. https://github.com/microsoft/autogen.

[17] Microsoft. 2023. AutoGen documentation. https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/quickstart.html.

[18] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. ChatDev: Communicative agents for software development. In *Proc. of 62nd Annual Meeting of the Association for Computational Linguistics*, Vol. 1. Association for Computational Linguistics, 15174–15186. doi:10.18653/v1/2024.acl-long.810

[19] Gabriele Russo Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum. In *Proc. of 2023 IEEE International Conference on Pervasive Computing and Communications, PerCom '23*. 131–140. doi:10.1109/PERCOM56429.2023.10099372

[20] Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. 2026. AI agents vs. agentic AI: A conceptual taxonomy, applications and challenges. *Information Fusion* 126 (2026), 103599. doi:10.1016/j.inffus.2025.103599

[21] Minghe Wang, Tobias Pfandzelter, Trever Schirmer, and David Bermbach. 2025. LLM4FaaS: No-code application development using LLMs and FaaS. In *Proc. of 18th IEEE/ACM International Conference on Utility and Cloud Computing, UCC '25*. ACM, Article 44, 8 pages. doi:10.1145/3773274.3774686

[22] Sebastian Werner, Mathis Kähler, and Alireza Hakamian. 2025. Code once, run green: Automated green code translation in serverless computing. In *Proc. of 2025 IEEE International Conference on Cloud Engineering, IC2E '25*. IEEE, 105–113.

[23] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025). doi:10.48550/arXiv.2505.09388

[24] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proc. of 61st Annual Meeting of the Association for Computational Linguistics*, Vol. 1. Association for Computational Linguistics, 769–787. doi:10.18653/v1/2023.acl-long.45

[25] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics, ACL '24*. Association for Computational Linguistics, 12834–12859. doi:10.18653/v1/2024.findings-acl.762