Latest updates: https://dl.acm.org/doi/10.1145/3758326

PDF Download
3758326.pdf
12 February 2026
Total Citations: 1
Total Downloads: 416

Published: 19 November 2025
Online AM: 05 August 2025
Accepted: 07 July 2025
Revised: 30 May 2025
Received: 04 March 2025

Citation in BibTeX format

RESEARCH-ARTICLE

A Language-based Approach to Macroprogramming for IoT Systems through Large Language Models

GIANLUCA AGUZZI, University of Bologna, Bologna, BO, Italy

NICOLAS FARABEGOLI, University of Bologna, Bologna, BO, Italy

MIRKO VIROLI, University of Bologna, Bologna, BO, Italy

Open Access Support provided by:

University of Bologna

A Language-Based Approach to Macroprogramming for IoT Systems through Large Language Models

GIANLUCA AGUZZI, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Italy
NICOLAS FARABEGOLI, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Italy
MIRKO VIROLI, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Italy

Large language models (LLMs) have transformed software engineering, particularly in code generation, where they assist developers in writing functions or entire programs. However, code generation remains challenging when the target domain is complex, as is the case with Internet of Things (IoT) systems. The challenge lies in capturing the entire system behavior within a *single* specification. developers often model only a subset of the system's functionality, focusing primarily on individual device behavior or data processing aspects, which may not address the core challenges of IoT, such as large-scale distributed coordination and emergent behavior. To address this, macroprogramming paradigms have been proposed as a means to specify the collective behavior of IoT systems more holistically. Among these approaches, aggregate computing stands out for its ability to express system-wide properties through a top-down, global-to-local perspective. Despite its potential, the adoption of aggregate computing remains limited due to the complexity of writing and maintaining such programs. To overcome these barriers, we propose a language-based approach based on macroprogramming that leverages LLMs for IoT code generation. Specifically, we employ the in-context learning capabilities of LLMs, guiding them to generate code based on an aggregate computing abstraction. This creates code that reflects system-wide properties and frees programmers from writing low-level code by letting them specify desired global properties in natural language. The LLM then translates these specifications into executable code, thus facilitating the development of collective intelligence applications in IoT systems.

CCS Concepts: • **Computing methodologies → Distributed programming languages**; *Cooperation and coordination*; • **Software and its engineering → Software system models**; **Software design engineering;**

Additional Key Words and Phrases: Large language models, macroprogramming, internet of things, aggregate computing, code generation

ACM Reference Format:
Gianluca Aguzzi, Nicolas Farabegoli, and Mirko Viroli. 2025. A Language-Based Approach to Macroprogramming for IoT Systems through Large Language Models. *ACM Trans. Internet Things* 6, 4, Article 26 (November 2025), 30 pages. https://doi.org/10.1145/3758326

This work is supported by the Italian PRIN project "CommonWears" (2020 HCWWLP), by the Gemma 2 Academic Award 2024, and by the FAIR foundation, funded by the European Commission under the NextGenerationEU programme (PNRR, M4C2, Investimento 1.3, Partenariato Esteso PE00000013, Spoke 8 "Pervasive AI").

Author's Contact Information: Gianluca Aguzzi, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Forlì-Cesena, Italy; e-mail: gianluca.aguzzi@unibo.it; Nicolas Farabegoli, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Forlì-Cesena, Italy; e-mail: nicolas.farabegoli@unibo.it; Mirko Viroli, DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Forlì-Cesena, Italy; e-mail: mirko.viroli@unibo.it.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 2577-6207/2025/11-ART26
https://doi.org/10.1145/3758326

1 Introduction

Research Context. With the advent of LLMs, the field of software engineering is undergoing a profound transformation, revolutionizing the way developers write code and develop software systems. Several code assistants, such as GitHub Copilot [61], have already demonstrated LLMs' potential in code generation [34], offering developers powerful tools to create code snippets, functions, or even entire programs [32]. Indeed, software engineering researchers are increasingly exploring the impact of LLMs on the field,[1] including testing [33], vulnerability detection [65], and multi-agent systems [31]. This progression from software engineering 1.0, where code is written from scratch, through 2.0 leveraging deep learning for specialized tasks, up to 3.0, where developers guide AI-driven code generation [30] is reshaping the software development landscape.

Research Gap. Among the various software engineering domains, the **Internet of Things** (**IoT**) is particularly challenging for code generation, especially in large-scale, interconnected systems, due to their distributed nature (*spatiality*), the large variety of devices (*heterogeneity*), and the complex interactions among them. Historically, IoT systems have been developed using traditional programming approaches, with the focus on individual devices and data processing aspects. These methods start to fall short when dealing with the *collective intelligence* of interconnected systems, as highlighted by recent roadmaps [15]. Particularly, IoT systems may be seen as **Collective Cyber-Physical Ecosystems** (**CCPEs**) [15], where the development focus is placed on the collective rather than the individual. In handling these CPPEs, the developers should consider both *time* and *space*, leading to *spatio-temporal* abstractions that are difficult to express in traditional programming languages. Moreover, the developers must account for unique factors such as emergent *system-wide behaviors* (e.g., moving patterns in a smart city), *simulations* (needed to test the system before deployment), and the tough problem of *global-to-local* mapping (i.e., how to translate global behaviors into local ones). Although literature has proposed multiple solutions, including macroprogramming [14] and automated methods [13], they still pose a barrier that separates practitioners from the complexities of writing and maintaining macroprograms. Existing research has applied LLMs to IoT (e.g., smart home orchestration [18], edge AI [42], and monitoring industrial IoT [37]), but little work explores their role in collective IoT programming. This gap highlights an opportunity to leverage LLMs in order to *comprehend* and *generate* code that reflects the collective intelligence of IoT systems.

Contribution. We introduce a language-based approach that uses LLMs to automate macroprogramming for large-scale IoT systems, explicitly targeting aggregate computing, a reference programming model. We select this paradigm since it is *space-time universal* [4], capable of expressing a wide range of collective behaviors in IoT systems, positioning it as a potential *lingua franca* for macroprogramming. The method supplies a hierarchy of composable abstractions, from basic field-calculus operators to advanced self-organizing patterns, enabling LLMs to progressively learn and generate code expressing collective behaviors. The solution co-designs a macroprogramming LLM agent capable of translating global specifications into executable local behavior. This leverages *in-context* learning within a human-in-the-loop framework, where the LLM is equipped with a carefully crafted minimal **body of knowledge** (**BoK**). When provided with natural language descriptions of desired system-wide behaviors, the LLM interprets these specifications and generates corresponding collective code based on aggregate computing abstractions. Human experts then evaluate the generated code, provide targeted feedback using specialized prompts when the code does not respect the expectations, and iteratively refine the BoK to address conceptual gaps or

[1] https://conf.researchr.org/home/2030-se

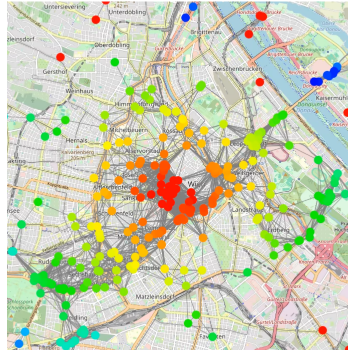


Fig. 1. Example of crowd density detection system for a large city event. Red areas indicate overcrowded zones.

misunderstandings. To ensure reliability, the pipeline employs unit tests that validate the code against the desired system-wide properties.

In presenting this work, we aim at replying to the following research questions:

RQ1 Are state-of-the-art LLMs capable of generating macro-code without a BoK?
RQ2 Are prompt engineering techniques sufficient to convey the abstractions of a macro paradigm?
- **RQ2.1** Can macroprogramming LLM agents understand the concept of "temporality"?
- **RQ2.2** Can macroprogramming LLM agents understand the concept of "spatiality"?
- **RQ2.3** Can macroprogramming LLM agents understand the concept of "space-time"?

RQ3 Given RQ2, are these LLMs able to generate novel/unseen solutions to macroprogramming problems using at least one macroprogramming language?

Paper Structure. The remainder of this article is structured as follows: Section 2 presents a motivating example to illustrate the challenges of programming large-scale IoT systems. Section 3 provides the theoretical foundation by examining IoT systems, macroprogramming paradigms with a focus on aggregate computing, and an overview of large language models with their application to IoT systems. Section 4 introduces our language-based approach for IoT macroprogramming through LLMs, detailing the co-design process, body of knowledge creation, and the human-in-the-loop framework. Section 5 presents a comprehensive evaluation of our approach, addressing each research question through both quantitative metrics and qualitative analysis of the LLM-generated code, including validation in a real-world scenario. Section 7 concludes by summarizing our contributions, discussing limitations, and outlining promising directions for future research.

2 Motivating Example

Consider a large city event (like a marathon in a European capital [11, 63]) where hundreds of participants attend the event, each equipped with a smartphone or a wearable device. The event organizers want to monitor the crowd density in real-time to ensure safety and security (see Figure 1). In particular, based on the density of people in a specific area, the organizers want to trigger different actions, such as sending alerts to security personnel to promptly intervene in case of overcrowding, or notifying attendees about the availability of nearby facilities. This app can be used by the participants also in case someone needs help because of a medical issue, reporting to the organizers the spot where the person is located, and allowing them to take the necessary actions. The participants, are supposed to have installed a mobile application providing information about the event, but also allowing the organizers to have a detailed view of the system in real-time.

The application is designed to run on many devices at the same time, where people can move freely, leave or enter the event area.

The application leverages the smartphones (or wearables) Bluetooth capabilities to detect the presence of other devices in the proximity, and possibly estimate the distance between them. Even though the GPS can be used to precisely locate the devices, its adoption can imply high costs in terms of battery consumption, but also in terms of privacy. Using the devices **Bluetooth Low Energy** (**BLE**) capabilities, the application can estimate how many devices are in the proximity, and the distance between them. Based on this collected data, the system can estimate the density of people in the event area, and when an overcrowded area is detected, this information is reported to the organizers, who can take consequent actions.

Implementing the crowding detection system using traditional programming paradigms can be challenging. Crowding detection using cameras is a well-known problem, and many solutions have been proposed in the literature [20, 21]. However, this implies the installation of cameras in the event area, which can be expensive and intrusive, making it difficult to deploy in a real-world scenario.

Capturing this collective and dynamic behavior is challenging and requires a modern approach: instead of reasoning in terms of individual devices, the designer can shape the system in terms of global behavior, where the system is programmed as a single unified entity. This lifts the abstraction level, allowing the designer to focus on system-wise properties. However, this abstraction shift may introduce non-trivial challenges for the designers, who may not be familiar with the programming paradigms that allow to express such collective behaviors. For this reason, providing non-expert designers with some support in writing code is crucial.

3 Background and Motivation

3.1 IoT Systems

The ubiquity of computation has long been a subject of interest, initially inspired by Mark Weiser's visionary ideas [58], and eventually evolving into what we now call the IoT. Defining IoT is inherently challenging, as it encompasses a wide range of systems and applications. IoT systems rely on *device connectivity* and *smart data processing*, using diverse communication protocols (e.g., Zigbee, Bluetooth, LoRa). Another defining characteristic of IoT is its *scale*; such systems often comprise many heterogeneous devices distributed across extensive geographic areas.

Building on Weiser's original vision, various extended perspectives have emerged over time, from pervasive computing [51] to edge/fog computing [3, 52] and, more recently, collective computing [1]. Among these perspectives, three key components are essential to understanding IoT systems: *Things*, *Runtimes*, and *Middleware*. *Things* represent the physical devices or entities within the IoT system that are equipped with sensors and actuators, enabling them to interact with their environment, collect data, and perform actions. These devices range from simple sensors like temperature monitors to complex systems like industrial robots [9]. These physical entities rely on *Runtimes*, the software components responsible for executing the application logic within the IoT system, including rule engines, data processing pipelines, and control algorithms that dictate how data collected by the things are processed, analyzed, and acted upon. The seamless integration between things and runtimes is enabled by *Middleware* components that act as intermediaries, facilitating communication and data exchange between the diverse devices and software elements in an IoT system by handling tasks such as device discovery, data routing, protocol translation, and security through popular solutions like message queues (e.g., MQTT, CoAP), API gateways, and data aggregation services [19].

Challenges in IoT Systems. IoT systems present numerous complexities that make them challenging to develop, deploy, and maintain. These challenges stem from several inherent characteristics

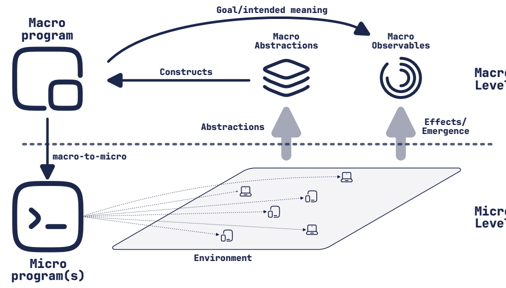


Fig. 2. Macroprogramming abstractions. Figure adapted from Reference [14].

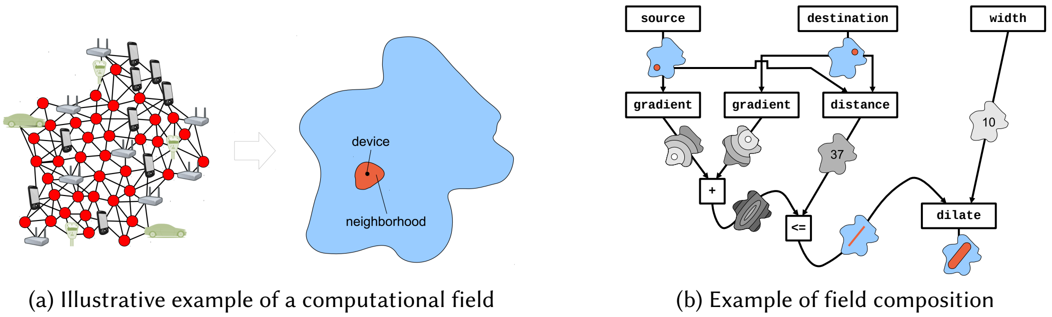
of IoT ecosystems: the extensive heterogeneity of devices with varying hardware specifications, computational capabilities, and power constraints; the diversity of communication protocols and networking technologies; and the distributed nature of computation across multiple physical locations. This heterogeneity extends to the *syntactic* and *semantic* levels, where variability in data formats and a lack of common understanding of data meaning can hinder effective communication. Furthermore, interoperability challenges emerge from the need for different systems to work together seamlessly, often requiring users to have specialized knowledge of each unit's unique characteristics. The distributed nature of IoT systems also introduces difficulties in terms of fault tolerance, scalability, and security, as ensuring reliable data exchange and protecting sensitive information across multiple devices and networks can be particularly challenging. Therefore, in this intricate scenario, finding the right programming paradigm and tools to manage the complexity of IoT systems is crucial for their successful development and deployment.

Programming IoT Systems. Throughout the years, many solutions have been suggested to tackle the difficulties associated with programming IoT systems [28]. These range from **general-purpose languages** (**GPLs**) for low-level device programming and cloud-based data processing, to **Model-Driven Engineering** (**MDE**) tools like ThingML [29] and Midgar [26] that enable high-level model creation with automatic code generation. Mashup tools like Node-RED address semantic heterogeneity by allowing users to interconnect pre-existing nodes representing devices and services, while End-User Development solutions such as IFTTT and Mozart [38] bridge the gap between system complexity and non-expert users. Additionally, domain-specific platforms have emerged for specific sectors, including home automation (openHAB, Jeedom) and smart agriculture (CropX, SWAMP [35]).

Despite these developments, an essential aspect remains noticeably lacking in the existing landscape. Traditional methods often take either a *device-centric* (namely, focusing on individual devices and their functionalities) or *data-centric* approach (namely, focusing on data processing and analysis), rather than considering the *collective behavior* of interconnected systems. We contend that there is a need for a collective-centric framework—one that encompasses the complex interactions among numerous "things." Macroprogramming, which focuses on deriving the overall system behavior from local interactions, presents a promising path to bridge this significant gap.

3.2 Macroprogramming

In essence, macroprogramming typically refers to a programming paradigm which has a form on *collective* (macro) abstractions as a first citizen in the programming model—see Figure 2 for an overview. This concept has its roots in the early 2000s, primarily within the **Wireless Sensor and Actuator Networks** (**WSAN**) domain [27]. Early works like TinyDB [41] (which provided



(a) Illustrative example of a computational field (b) Example of field composition

Fig. 3. Aggregate computing: Devices interact via computational fields that abstract device count and evolve over time (a), enabling macro-operations to specify global behavior while local computations emerge automatically (b).

a SQL-like interface for WSNs), Pieces [40] (focusing on composing software components for pervasive systems), Abstract Regions [59] (for defining logical groupings of nodes), and Regiment [45] (offering a stream-processing abstraction over sensor networks) explored approaches for specifying macro-system behavior, abstracting away from low-level details like message passing and distributed computation. Other contemporary approaches for WSNs included Kairos [27] (a data-centric paradigm similar in goals to TinyDB). These foundational works have since expanded, finding applications in diverse areas including spatial computing [10], swarm robotics (e.g., Buzz [13], a DSL for robot swarms), and **Collective Adaptive Systems** (**CAS**) engineering [25] (e.g., SmartSociety [53], which models CAS through aggregate programming constructs for simulation). This view is then enforced by recent work on CCPEs where the authors pose the need for a new engineering discipline that focuses on the collective behavior of systems [15].

Macroprogramming is particularly well-suited for IoT systems due to its ability to tackle the inherent complexities of heterogeneity, scale, and distribution. By specifying behavior at the collective level, developers are freed from intricate low-level details. In fact, several macroprogramming paradigms have been proposed to address the challenges of programming IoT systems, including PyoT [8] (a Python-based framework for general IoT), Warble [50] (a framework tailored for smart IoT applications), and many others [14] such as DDFlow (a DSL for data-centric stream processing). Among these paradigms, this article focuses on *aggregate computing*, presented in the next section, known for its high abstraction complexity, strong scalability, and robust approach to general distributed systems.

3.2.1 Aggregate Computing. A recent instantiation of macroprogramming is aggregate computing [11], which builds on *computational fields* [43]—By manipulating these fields through macro-level operations, developers specify system-wide behavior while low-level local computations are derived automatically (see Figure 3 for an overview).

Aggregate computing offers several advantages for IoT applications. In particular, the behavior scales naturally with the number of devices because the primary abstraction is the field, which inherently abstracts from device count. Furthermore, this paradigm identifies several key properties, such as *self-stabilization* [55] and *eventual consistency* [12], that demonstrate the system's robustness in the face of reaching a stable state despite transient faults. Moreover, aggregate computing is proven to be *space-time universal*, meaning that any computable function can be expressed in this paradigm [11].

Another key feature of the paradigm is its deployment independence [56], namely, the ability to express the same *functional* behavior in several architectural styles, ranging from centralized

to edge-cloud to fully distributed. This capability is supported by recent discoveries in pulverized architectures [23, 24], which affirm that such independence is inherent to deployment.

Aggregate computing follows a layered approach, with the core constructs rooted in *field calculus* [7], then introducing *self-organizing building blocks* [16] that enable the expression of complex collective behaviors in a composable way. Field calculus [7] which serves as a foundational model for aggregate computing, providing a set of core constructs for expressing field manipulations. For a full understanding of the field calculus, we refer the reader to the original work [7]. In the following, we will introduce the key building blocks of field calculus via examples leveraging ScaFi [16], a Scala-based library for aggregate computing as well as its computational model to understand how the global stance of aggregate computing is achieved.

Computational Model. In this model, a set of devices is deployed in a spatial environment, where each device is equipped with sensors, actuators, and communication capabilities. Each device uses its local state along with information received from its neighbors to compute its next state. The aggregate computation paradigm follows a self-organizing, round-based evaluation process, where devices continually execute the macroprogram, update their state, and propagate relevant information, thereby enabling emergent, collective behavior.

Each evaluation round comprises three principal phases: *context acquisition*, *program evaluation*, and *environment update*. In the first phase, each device collects its local context, which consists of messages received from neighboring devices, its previous state, and data obtained from onboard sensors. Utilizing this context, the device evaluates the macroprogram—a stateful function that processes the input context and produces an *export*. This export encapsulates the data that the device intends to share with its neighbors in the subsequent round. In the environment update phase, the device updates its internal state, transmits the export to its neighbors, and performs necessary actions on its actuators (e.g., activating a light).

This iterative round-based evaluation continues until the system achieves the global objective specified by the macroprogram. Moreover, this continuous evaluation process exhibits resilience to transient faults, ensuring that the system eventually converges to a stable state where no further changes occur.

Field Calculus in a Nutshell. Field calculus APIs provide fundamental constructs for manipulating computational fields, enabling field creation, transformation, and aggregation. Different implementations like FCCP [6] or Protelis [48] expose these constructs through varied conventions. As shown in Table 1, the core field calculus operators are as follows. sense retrieves a named sensor's data to build a field (for example, temperature readings); rep models temporal evolution by maintaining and updating local state across rounds; nbr allows a device to access the value of an expression as evaluated at each neighbor; foldhood aggregates those neighbor-accessed values using a binary operator (e.g., sum, min, max); and conditional logic is provided by branch, which partitions the network into non-communicating regions, and mux, which simply selects between two values without creating partitions. Spatio-temporal abstractions typically combine these operators. For instance, nesting foldhood within rep can create fields that converge to global properties over time. For example, the following code computes the maximum perceived temperature of the system:

```
1 rep(Double.NegativeInfinity) { temp => maxHood { nbr(sense("temperature")) } }
```

3.2.2 Self-Organizing Building Blocks. Building on the core constructs, aggregate computing provides self-stabilizing building blocks that guarantee convergence to stable configurations despite network perturbations. As shown in Table 2, aggregate computing relies on three fundamental self-stabilizing building blocks: *gradient cast* (G) propagates values outward from source nodes by

Table 1. Field Calculus API with Signatures and Semantics

API	Signature	Semantics
sense	`sense[A](name: String):A`	Retrieves data from a named sensor
rep	`rep[A](init: A)(evolve: (A)=> A):A`	Creates temporal evolution with initial value and evolution function
nbr	`nbr[A](expr: => A): A`	Accesses values from neighboring devices
foldhood	`foldhood[A](init: A)(acc: (A,A)=> A)(exp: => A):A`	Aggregates values from neighbors using a binary operation
min,max,avg/Hood	–	Specific variations of foldhood for min/-max/average operations
branch	`branch[A](cnd: => Boolean)(th: => A)(el: => A):A`	Partitions the system based on a condition (creates non-communicating parts)
mux	`mux[A](cnd: => Boolean)(th: A)(el: A):A`	Selects between two values based on a condition (without partitioning)

Table 2. Summary of Self-Stabilizing Building Blocks in ScaFi

Building Block	Signature	Purpose
Gradient Cast (G)	`G[A](source:Boolean,local:A,accumulation:A=>A,metric:()=>Double):A`	Creates a potential field from source nodes and propagates values outward
Collect Cast (C)	`C[A](potential:Double,aggregation:(A,A)=>A,local:A,Null:A):A`	Aggregates values from nodes towards a sink following a potential field
Sparse Choice (S)	`S[A](influenceArea:Double,metric:()=>Double):Boolean`	Partitions the system into regions with leader nodes

combining neighbor metrics with a user-defined accumulation function; *collect cast* (C) follows these gradients to aggregate local values toward one or more sink nodes; and *sparse choice* (S) partitions the network into non-overlapping regions by electing leader nodes within specified influence zones. These building blocks address limitations in naive implementations like the max temperature example, which converges to the maximum temperature *ever* reported rather than adapting to changes. Moreover, these operators enable composition of complex self-stabilizing patterns:

```
val leader = S(2.0, nbrRange)
val potential = G(leader, 0.0, _ + nbrRange(), nbrRange)
val globalTemperature = C(potential, _ + _, sense("temperature"), 0)
val nodes = C(potential, _ + _, 1, 0)
G(leader, globalTemperature / nodes, value => value, nbrRange)
```

This composition first uses `S` to elect leader nodes, effectively partitioning the network into regions where each leader has an influence area defined by a radius of `2.0` distance units (using `nbrRange` as the metric). Then, `G` creates a potential field (i.e., a field or distance with respect to a source zone) originating from each leader, establishing paths within these regions. Following this, two `C` operations are used: one sums the `sense("temperature")` from all devices within each leader's region (following the `potential`), and the other counts the number of nodes in each region. The division of these two collected values yields the average temperature for each leader's region. This average temperature is then broadcast by each leader to all nodes within its respective region using the final `G` operation.

3.3 Large Language Models—Primer

This section introduces the concept of LLMs from a *user perspective*, providing an overview of their capabilities and applications, though readers seeking more in-depth technical discussion should consult comprehensive overviews available in the literature [44]. In the literature, a *language model* is defined as a statistical framework that assigns probabilities to word sequences by predicting the subsequent word based on the preceding context. Over time, language models have progressed from simple n-gram approaches to sophisticated neural network-based architectures. In particular, transformer-based models [54] have revolutionized natural language processing by enabling the creation of large-scale models with billions of parameters, thanks to their self-attention mechanism that captures long-range dependencies in text data and their parallelizable architecture that accelerates training. The term "large" in LLMs refers to both the model's extensive parameter count (billions of weights) and the vast datasets (trillions of examples) used for training. These models are typically trained through self-supervised learning, where they learn to predict subsequent words based solely on preceding context. This approach represents a middle ground between unsupervised and supervised learning, as the model generates its own supervision signal from input data without requiring explicit human-provided labels.

Once trained on massive corpora of text data, these models emerge as versatile *foundational* systems capable of understanding and generating human-like text across diverse domains. These foundation models serve as versatile starting points that can be either fine-tuned for specific downstream tasks or leveraged directly through *in-context learning* techniques (more later). Indeed, despite variations in specific architectures (e.g., mixture of experts), these advanced models share the core principle, namely the use of a *prompt* (i.e., a user-defined input sequence) to guide the generation of task-specific text. In this instrumentation, LLMs demonstrate remarkable versatility across a wide range of tasks [36]. This adaptability is achieved through *prompt engineering* [17], which involves the careful design of prompts to direct the model toward desired outputs. When a model is provided with a prompt that consists solely of a sequence of instructions or ideas, it performs *zero-shot* [49] learning (or *role-playing*), adapting its responses based only on the provided context. Conversely, including examples or detailed instructions within the prompt enables *in-context* or *few-shot* learning [57], allowing the model to effectively "learn" new tasks from the input.

3.4 Large Language Models for IoT Systems

The adaptability of LLMs to diverse contexts has prompted extensive research into their integration with IoT systems. Recent studies have explored multiple directions [60], including leveraging LLMs for sensor data analysis, automated code generation for IoT devices, and high-level reasoning and planning for IoT ecosystems [42]. Several researchers have investigated the fundamental components required for effective deployment of generative models in IoT environments, coining the term "Generative IoT" to describe this emerging paradigm [60]. This convergence aims to address challenges in scalability, contextual understanding, and human-machine interaction within distributed systems. Notable frameworks in this domain include CASIT [64], which provides a mechanism for translating sensorial data into textual representations, enabling collaboration between multiple LLMs for analyzing environmental data and responding to user queries. Similarly, LLMind [18] integrates various AI models to facilitate cooperation among IoT devices, with the LLM acting as an orchestrator that mediates communication between physical devices and human operators, thereby enhancing the system's capability to execute complex tasks in heterogeneous environments.

Despite these advances, current LLM integration approaches for IoT systems face several critical limitations, including tendency toward hallucinations and constrained reasoning capabilities

Table 3. Legend: H = High; M = Medium; L = Low; P = Platform-Dependent; ✓ = Yes / High; ~ = Partial / Medium; ×= No / Low

Paradigm	Abstraction Complexity	Scalability	Robustness	Perspective	Expressiveness	Universality	Primary Domain
Aggregate Computing [11]	H	H	✓	Global	DSL	✓	General
Buzz [13]	M	M	~	Local	DSL	×	Swarm Robotics
PyoT [8]	M	P	~	Data	Framework	×	General IoT
Kairos [27]	M	M	×	Data	Framework	×	Wireless Sensor Networks
DDFlow [14]	H	H	~	Data	DSL	×	Data/Stream
Warble [50]	M	P	~	Local	Framework	×	Smart IoT
SmartSociety [53]	H	M	~	Global	DSL	×	Collective Adaptive Systems
Regiment [45]	M	M	×	Global	DSL	×	Wireless Sensor Networks
TinyDB [41]	H	M	×	Data	DSL	×	Wireless Sensor Networks

regarding spatio-temporal dynamics. An alternative approach to direct orchestration involves leveraging LLMs for code generation in IoT contexts. Recent research [66] has investigated the capacity of LLMs to generate macroprogramming code for IoT systems leveraging PyoT [8], strategically reducing the reasoning burden on generative models by employing high-level abstractions to guide the code synthesis process.

3.5 Motivation

This article aims to explore the convergence of two significant technological trends: on one hand, the emergence of versatile large language models with unprecedented capabilities, and on the other hand, a paradigm shift toward collective system programming rather than focusing on individual devices. Our work is motivated by recent promising applications combining macroprogramming approaches and LLMs, and recent language-based approaches [2], which demonstrate the potential for addressing the complexity of distributed IoT systems through higher-level abstractions.

To effectively harness LLMs for this paradigm shift, a robust and expressive macroprogramming foundation is essential. Among the various paradigms (as summarized in Table 3), we selected aggregate computing [11] as the foundational approach for this research. This paradigm offers a compelling **global-to-local perspective**, allowing LLMs to specify system-wide behaviors through operations on *computational fields*. This top-down approach is crucial for managing complexity and is well-suited for translation from high-level, natural language specifications that an LLM can process. Furthermore, its provides strong theoretical underpinnings and practical advantages: it exhibits high **scalability** and inherent **robustness** (including self-stabilization properties [55]), and supports **deployment independence** [56]. Critically, the paradigm is proven to be **space-time universal** [11] (Table 3, Universality: ✓). This universality is paramount, as it positions this paradigm not merely as one option among many, but as a potential *lingua franca* for macroprogramming—a foundational model capable of expressing a vast range of collective behaviors and potentially underpinning or relating to other macroprogramming concepts. This makes it an ideal, generalizable target for LLM-based code generation.

4 Methods

In this article, we present a pipeline that leverages prompt engineering and *in-context learning* (or *few-shot learning*) to enable LLMs to automatically generate macroprogramming code for IoT systems, specifically targeting the aggregate computing paradigm. To provide a concrete test bed, we adopt ScaFi [16], a Scala-based DSL that offers a direct mapping to field-calculus operators and self-stabilizing building blocks. ScaFi's close alignment with the theoretical model, along with its seamless integration with large-scale simulation environments such as Alchemist [47], makes it an ideal platform for validating LLM-generated collective programs and ensuring that the intended spatio-temporal and convergence properties of aggregate computing are preserved.

Our approach supplies the LLM with a carefully curated set of illustrative examples and then prompts it to synthesize code for new tasks, guiding the model to recall or rediscover the core abstractions of aggregate computing (e.g., rep, nbr, foldhood).

The solution is divided into two main phases: an *adaptation* phase, where we create an adapted version of an LLM capable of generating code in our reference language, and a *deployment* phase, where practitioners unfamiliar with specialized programming paradigms can develop CASs using natural language specifications.

4.1 Adapatation Phase

As illustrated on the left side of Figure 4, the first phase involves transferring domain knowledge from human experts to the LLM agent. This knowledge extraction and formalization process consists of three key stages, *BoK definition*, *Test case definition*, and *Co-design phase*. Finally, a clear *input-output* mapping is established to facilitate the LLM's understanding of the task at hand.

Definition of the BoK:. This stage involves creating a concise operational manual to help LLMs understand (or recall from their training dataset) the principal abstractions. The BoK is structured as a progressive sequence of executable examples that introduce ScaFi's API elements incrementally, starting with basic operators like rep and sense, then advancing to spatial operators like nbr and foldhood, and finally to self-organizing building blocks. Each example demonstrates concrete API usage rather than abstract concepts, allowing the LLM to learn from code patterns and their applications in increasing complexity. The general structure for each example is as follows:

```
# SCENARIO_DESCRIPTION #
# API_INTRODUCTION #
# API_USAGE #
```

For example, the BoK entry for temporal evolution is structured as follows:

```
# Temporal Evolution allows devices to maintain and update state over time #

# The rep construct allows stateful computation:
def rep[A](init: => A)(evolve: A => A): A
- init: The initial value
- evolve: Function that takes the current value and returns the next value

# Example: Creating a counter that increases by 1 each round
def main(): Int = rep(0)(_ + 1)
When executed repeatedly:
Round 1: 0
Round 2: 1
Round 3: 2
```

For the complete BoK, please refer to the public repository.[2]

Test case Definition. Each test case is structured to evaluate the LLM's capability to generate correct aggregate computing code and is composed of four main components:

- **Test description:** A natural language description of the expected behavior of the program (e.g., "Compute the average temperature of the system"), which serves as the prompt provided

[2]https://github.com/nicolasfara/experiments-2025-acm-iot-ac-llm

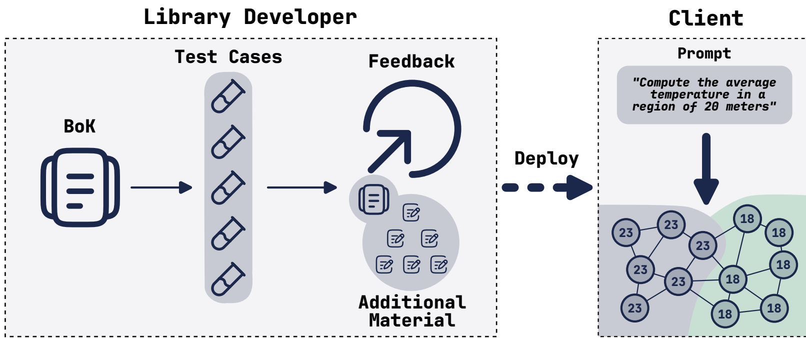


Fig. 4. The proposed pipeline to co-generate an LLM agent which is able to generate code which understands collective abstraction and can be used to program IoT systems.

to the LLM for code generation. These descriptions are deliberately crafted with varying levels of complexity to test different aspects of spatio-temporal reasoning.
- **Ground Truth:** A reference program that correctly implements the expected behavior, developed by aggregate computing experts. This serves as the baseline for functional correctness and is used to verify the assertions when testing the generated code.
- **Initial State:** The configuration used to initialize the simulation environment, represented as a network graph where nodes correspond to devices and edges represent communication links. Each node is assigned specific properties (e.g., position coordinates, sensor values, device IDs) that are necessary for the test execution.
- **Assertions:** A set of programmatic conditions that verify whether the generated code exhibits the correct behavior. These assertions check both local properties (e.g., specific node values) and emergent global properties (e.g., convergence patterns, collective behavior). For spatio-temporal tests, assertions verify the system behavior after sufficient rounds to achieve stabilization.

Co-design phase: In this phase, human experts iteratively refine the BoK based on the LLM's performance on test cases. After running a test, experts analyze the generated code to identify specific misunderstandings or missing concepts. They then update the BoK by adding targeted explanations, clarifications, or examples that address these gaps. This process is repeated: the updated BoK is provided to the LLM, new code is generated, and further adjustments are made as needed. While some automation is possible (e.g., using text-based differentiability systems [62]), expert oversight is essential to ensure that the BoK provides the right level of abstraction and does not simply give away the solution.

For the co-design phase, we crafted a template to guide the interaction between the human expert and the LLM. The template is as follows:

```
Knowledge: # BOK #
Given the knowledge,
Task: # PROMPT_DESCRIPTION #
Generated code: # CODE_GENERATED #
Expected code: # TEST_CASE #

What concepts were misunderstood?
```

Where the `PROMPT_DESCRIPTION` is the description of the task to be solved, the `CODE_GENERATED` is the code generated by the LLM, that is not correct, and the `TEST_CASE` is the expected code. Through several iterations of this phase, the BoK is progressively refined until the prompt reliably enables the LLM to generate correct and effective programs automatically.

Input and Output Decoding. To ensure reproducibility and methodological rigor, we implemented a standardized input/output protocol for the LLM-based code generation process. The protocol consists of a structured prompt template that combines the BoK, task description, and precise output formatting requirements (also known as *structured output*). The global prompt template follows this structure:

```
You are an expert in Aggregate Computing and the ScaFi programming language.
I will give you a programming task, and you should provide a solution in ScaFi.

Body of Knowledge:
{BOK}

TASK:
{TASK_DESCRIPTION}

FORMAT REQUIREMENTS:
- Return ONLY a single ScaFi function with signature "def main(): Any"
- Do not include explanations, comments, or additional text
- Ensure the code is syntactically valid and correctly implements the requested behavior

Your answer:
```

This template ensures consistent interaction with the LLM agent across all experiments. The `{BOK}` placeholder is populated with the appropriate knowledge level (none, basic, or with building blocks) during execution. The `{TASK_DESCRIPTION}` contains the natural language specification of the desired collective behavior.

The rigorous output requirements are essential for automated processing of LLM responses. By enforcing a standardized output format (`def main(): Any`), we reduce parsing ambiguities and enable direct compilation without intermediate processing steps.

4.2 Deployment and Operationalization Phase

The deployment phase constitutes the operationalization of the co-designed LLM agent for practical application. In contrast to black-box orchestration approaches described in related literature, our methodology maintains complete transparency of the generated programs, enabling verification and analysis of emergent system behaviors. During this phase, domain practitioners provide natural language specifications of desired collective behavior, which the LLM agent translates into executable aggregate computing code. The generated code undergoes a multi-stage validation pipeline:

1. **Static analysis:** The ScaFi compiler is used to check for syntactic correctness and type safety.
2. **Dynamic verification:** The code is executed in a controlled simulation environment to validate its functional properties and emergent behaviors.
3. **Deployment and monitoring:** The validated code is deployed to physical infrastructure, where it is instrumented for runtime monitoring to ensure correct operation and to detect potential issues during real-world execution. This step includes continuous observation and, if necessary, intervention to maintain system reliability.

This workflow ensures that generated programs exhibit both theoretical correctness and practical reliability in real-world distributed environments.

5 Evaluation

5.1 Setup

To exercise the approach proposed in Section 4, we set up a pipeline meant to test with different BoKs and LLMs the capability of generating code for IoT systems using aggregate computing. The pipeline provides a class meant to test a single program given a BoK, an LLM, a prompt describing the expected behavior, and a test-suite to evaluate the generated code. The test-suite is adapted from the ScaFi library,[3] which already provides utilities for verifying the correctness of ScaFi programs. Additionally, the ScaFi codebase provides a set of unit tests that are used to evaluate the LLMs generated code. Each test provides a baseline implementation of the expected behavior, acting as a control group to compare the generated code, and verify the correctness of the assertions. Each test is executed 20 times to ensure the stability of the results. The generated results are collected and analyzed to evaluate the performance of the LLMs in generating code for this IoT domain. The experiments are released as open-source software,[4] and permanent access is provided through Zenodo [22] with a permissive license for reproducibility and further research.

5.1.1 Real-World Application Evaluation. To validate our approach in a practical context, we conducted an empirical evaluation using the highest-performing LLM model (Gemini 2.5 Pro) to generate code for the crowding detection scenario described in Section 2. Specifically, we utilized mobility data collected from a city marathon dataset [11, 63].

The generated program was deployed in a controlled simulation environment using Alchemist [47], a simulation platform designed for evaluating pervasive computing and large-scale IoT scenarios. The simulation was configured with a network of mobile nodes representing event participants with wearable/mobile devices ($\sim$ 300), incorporating realistic movement patterns and Bluetooth signal propagation characteristics.

We evaluated the system's performance against three critical functional requirements: (1) qualitative in identifying crowded areas based on device density and proximity metrics, (2) effective propagation of alert signals within spatially-bounded affected regions, and (3) robustness to dynamic network topology changes induced by participant mobility.

5.1.2 Testing Framework. The testing framework is implemented in Scala 3 since it seamlessly integrates with the ScaFi library. We setup the `AbstractScafiTest` class to provide a common interface for the test cases. This class requires:

— A list of BoK files to provide with the LLM;
— The file containing the prompt to be used to generate the code with the LLM;
— The list of LLMs implementations to be tested; and
— The number of iterations should be run for each combination of BoK and LLM.

Additionally, when implementing a test case, the `baselineWorkingProgram` method must provide the expected behavior of the program, acting as a baseline to validate the test; the `programTests` method must provide the assertions meant to verify the correctness of the LLM generated code, namely the assertions to ensure the program correctness.

Once all the test cases are implemented (11 in total), the testing framework, executes all the test cases and collects the results. In particular, for each BoK and LLM combination, an API call

[3] https://github.com/scafi/scafi
[4] https://github.com/nicolasfara/experiments-2025-acm-iot-ac-llm

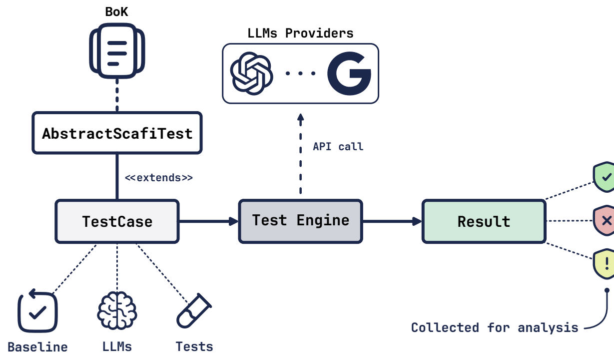


Fig. 5. Pictorial representation of the testing workflow.

is performed to the LLM API to generate the code. The generated output is collected and via a predefined template is compiled and executed. In the case a compilation error occurs, it is collected the BoK and LLM used, the produced code, and the error message marking the test as failed. In the case the code compiles and runs, the test suite is executed, and based on the test outcome, the test is marked as passed or failed.

With this setup, three possible outcomes can occur:

— `Success` if the code compiles and the test suite passes;
— `Error` if the generated code does not compile or an unexpected runtime error occurs; and
— `TestFail` if the code compiles but the test suite fails.

All produced outcomes are then collected and stored to conduct a meaningful analysis of the results. The Figure 5 provides a representation of the described testing workflow.

5.1.3 *Prompts and BoKs.* In the Table 4, we provide the list of implemented test cases, their respective prompts, and the type of test they represent. The test cases are divided into three main categories:

— *Temporal* or *Spatial*. Use of either temporal or spatial interactions;
— *Spatio-temporal*. Combination of both temporal and spatial interactions; and
— *Building Blocks*. Use of self-stabilizing building blocks.

This division provides a categorization of test cases based on their computational complexity and the aggregate computing abstraction they employ. Following the core principles of aggregate computing discussed in Section 3.2.1, we progressively increase complexity from simple temporal or spatial operators to their combination in spatio-temporal patterns, and finally to compositions of self-organizing building blocks.

The gradient computation establishes a potential field from a source node, producing the visual pattern shown in Figure 6(a). When obstacles are introduced (Figure 6(b)), the gradient must adapt to route around these impediments, requiring a more sophisticated understanding of spatial constraints and field propagation. Building upon gradient computations, the *channel* test cases (Figure 6(c) and 6(d)) create a path connecting source and destination nodes—an abstraction that exemplifies the composition of simpler field operations to achieve higher-level spatial structures.

Finally, the most complex test case involves area-wise temperature monitoring (Figure 6(e)), which requires partitioning the network into zones, computing aggregate properties within each zone, and distributing alerts when conditions exceed thresholds (see Section 3 for more details).

Table 4. List of Test Cases with their Respective Prompts and Types

Test Case	Prompt	Type
Count Down	`Evolve backwards a value from 1000 to 0`	*Temporal*
Neighbors Count	`Count neighbors`	*Spatial*
Neighbors Count Excluding Self	`Count neighbors excluding self`	*Spatial*
Gather Neighbors IDs	`Gather a list IDs of their neighbors`	*Spatial*
Calculate Min Distance Neighbors	`Calculate the min distance ONLY from neighbors`	*Spatial*
Collect Max ID	`Compute the max ID in the whole network`	*Spatio-temporal*
Calculate Gradient	`Compute the euclidean distance from the source using a sensor named ``source''`	*Spatio-temporal*
Calculate Gradient (Obstacles)	`Compute the euclidean distance from the source using a sensor named ``source'' and it should avoid any obstacles in the path (namely, boolean sensors called ``obstacle'').`	*Spatio-temporal*
Create Channel	`Create a channel from the source node called `source' to the destination node called `destination'`	*Building blocks*
Create Channel (Obstacles)	`Create a channel from the source node called `source' to the destination node called `destination' and it should avoid any obstacles in the path (namely, boolean sensors called `obstacle'). It should return true if the channel was successfully created, and false otherwise.`	*Building blocks*
Area-wise Alarm for Temperature Above 30	`Compute several areas where it is computed the area-wise temperature and send back (in broadcast) within the area an alarm to the area when the temperature is above 30 degrees.`	*Building blocks*

The background colors indicate the type of test case: *temporal or spatial* (green), *spatio-temporal* (orange), and *building blocks* (red).

5.2 Metrics

To evaluate the performance of LLMs in code generation tasks, we employ two metrics: **pass@k** and **compile@k** which are defined as follows: The former quantifies the probability that at least one successful solution exists among k independent generations for a given problem. This metric, established in the literature for evaluating code generation models [39], is formally defined as

$$\text{pass@k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \tag{1}$$

where n is the total number of attempts (20 in our case), and c is the number of successful solutions. This metric captures the likelihood of generating at least one correct solution within k attempts, providing a more comprehensive assessment of model performance than a simple success rate. As a complementary measure, we introduce compile@k, which follows the same statistical formulation but focuses on syntactic correctness rather than functional correctness:

$$\text{compile@k} = 1 - \frac{\binom{n-d}{k}}{\binom{n}{k}}, \tag{2}$$

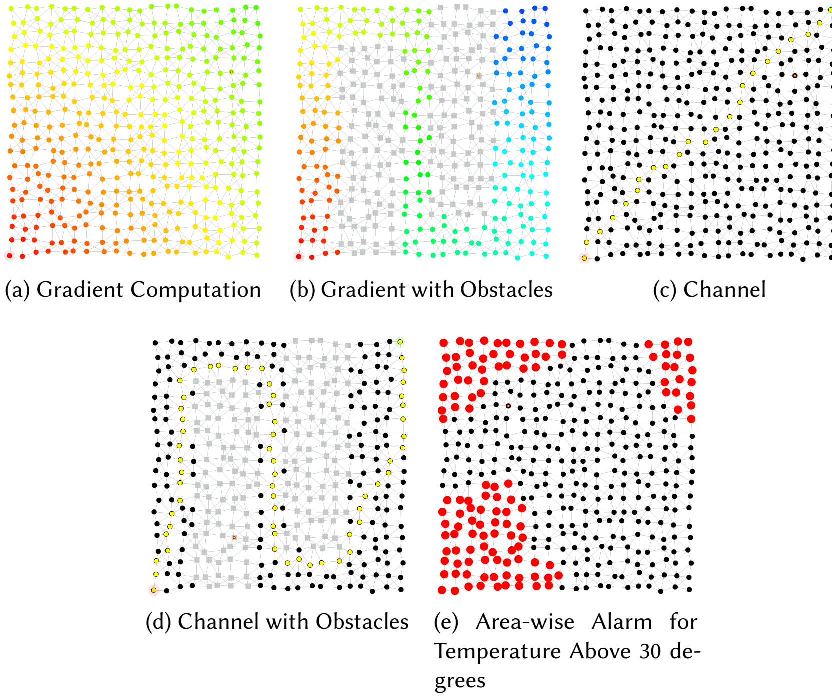

(a) Gradient Computation (b) Gradient with Obstacles (c) Channel (d) Channel with Obstacles (e) Area-wise Alarm for Temperature Above 30 degrees

Fig. 6. Screenshots of the program generated by the LLM under a simulated scenario with Alchemist [47]. In (a) and (b), the colors are proportional to the distance from the source node, where the red shadow indicates the source node (bottom left). In (b) and (d), the square gray nodes represent obstacles. In (c) and (d), the yellow nodes indicate if a node is part of the channel. Finally, in (e), the red nodes indicate the area where the temperature is above 30 degrees.

where d is the number of solutions that successfully compile. This metric evaluates the model's ability to generate syntactically valid code, complementing the functional correctness measured by pass@k.

In our evaluation, we report results for $k \in \{1, 5, 10\}$ to characterize performance across different generation budgets.

5.3 Results

Detailed metrics for all models across these knowledge levels are reported in Table 5 where the pass@k and compile@k metrics for each model are presented, giving a comprehensive overview of their performance.

5.3.1 Overview. Figure 7 illustrates the outcomes distributed across the test cases for each model. The top-performing model, *Gemini 2.5 Pro*, successfully generates code that both compiles and passes tests more than 50% of the time, while *Claude 3.7 Sonnet* (second performing model) achieves a pass rate of around 41%.

In general, a compilation error occurs in 20–30% of the cases, except for *Mistral 8b*, which has a significantly higher failure rate of around 60%. The overall trend indicates that larger models tend to perform better, except for *Llama 3.1 405B*, which does not perform as well as its parameters count might suggest. This may indicate that some models were trained on more aggregate computing data than others, therefore more effectively activating their latent capabilities.

Table 5. Pass@k (p@k) and Compile@k (c@k) Metrics by Model and Knowledge Level

	No Knowledge						Basic Knowledge						Knowledge+BB					
Model	**p@1**	**p@5**	**p@10**	**c@1**	**c@5**	**c@10**	**p@1**	**p@5**	**p@10**	**c@1**	**c@5**	**c@10**	**p@1**	**p@5**	**p@10**	**c@1**	**c@5**	**c@10**
Gemini 2.5 Pro	0.27†	0.47†	0.50†	0.39†	0.72†	0.77†	0.66†	0.73‡	0.73*	0.86†	0.96*	0.99*	0.64†	0.79‡	0.84‡	0.91†	1.00†	1.00†
Claude 3.7B Sonnet	0.02‡	0.10‡	0.17‡	0.15*	0.45*	0.62*	0.62‡	0.75†	0.80†	0.83‡	0.99†	1.00†	0.59*	0.74*	0.79*	0.90‡	0.98‡	1.00‡
Gemini 2.0 Flash Exp	0.01	0.04	0.07	0.01	0.04	0.07	0.60*	0.68*	0.71	0.75	0.90	0.91	0.60‡	0.80†	0.86†	0.88*	0.93*	0.95
Llama 3.1 405B Instruct	0.00	0.00	0.00	0.07	0.26	0.36	0.36	0.58	0.66	0.70	0.89	0.91	0.42	0.64	0.73	0.70	0.90	0.97
Llama 3.3 70B Instruct	0.00	0.00	0.00	0.01	0.06	0.11	0.48	0.67	0.75‡	0.75	0.91	0.97	0.45	0.58	0.61	0.59	0.75	0.80
GPT-4.1 Mini	0.02*	0.08*	0.13*	0.08	0.30	0.45	0.49	0.58	0.61	0.83*	0.96‡	0.99‡	0.50	0.59	0.61	0.75	0.87	0.94
Codestral	0.00	0.00	0.00	0.11	0.29	0.41	0.30	0.49	0.57	0.51	0.86	0.93	0.42	0.59	0.68	0.65	0.90	0.95
Qwen 2.5 Coder 32B Instruct	0.00	0.00	0.00	0.04	0.14	0.21	0.39	0.55	0.61	0.59	0.86	0.93	0.34	0.51	0.58	0.56	0.84	0.93
Mistral 3.1 24B Instruct	0.00	0.00	0.00	0.04	0.09	0.09	0.42	0.52	0.57	0.62	0.83	0.90	0.41	0.54	0.59	0.54	0.74	0.83
Llama 4 Maverick	0.00	0.00	0.00	0.00	0.02	0.05	0.52	0.55	0.55	0.77	0.87	0.90	0.39	0.45	0.50	0.68	0.76	0.80
Llama 4 Scout	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.46	0.50	0.63	0.89	0.97	0.29	0.46	0.54	0.63	0.93	0.98*
Gemini 1.5 Flash	0.00	0.00	0.00	0.00	0.00	0.00	0.55	0.55	0.55	0.62	0.64	0.64	0.45	0.45	0.45	0.72	0.79	0.81
Gemma 3.27B Instruct	0.00	0.00	0.00	0.00	0.00	0.00	0.27	0.27	0.27	0.65	0.77	0.82	0.38	0.49	0.57	0.50	0.67	0.75
Gemma 3.12B Instruct	0.00	0.00	0.00	0.02	0.10	0.15	0.38	0.45	0.45	0.62	0.82	0.86	0.30	0.36	0.36	0.55	0.78	0.86
Gemma 3.4B Instruct	0.00	0.00	0.00	0.35‡	0.65‡	0.75‡	0.14	0.18	0.18	0.69	0.81	0.82	0.09	0.09	0.09	0.76	0.90	0.91
Mistral 8B	0.00	0.00	0.00	0.10	0.30	0.42	0.00	0.00	0.00	0.55	0.84	0.90	0.00	0.02	0.05	0.55	0.84	0.90

† 1st, ‡ 2nd, and * 3rd best performance per metric and knowledge level.

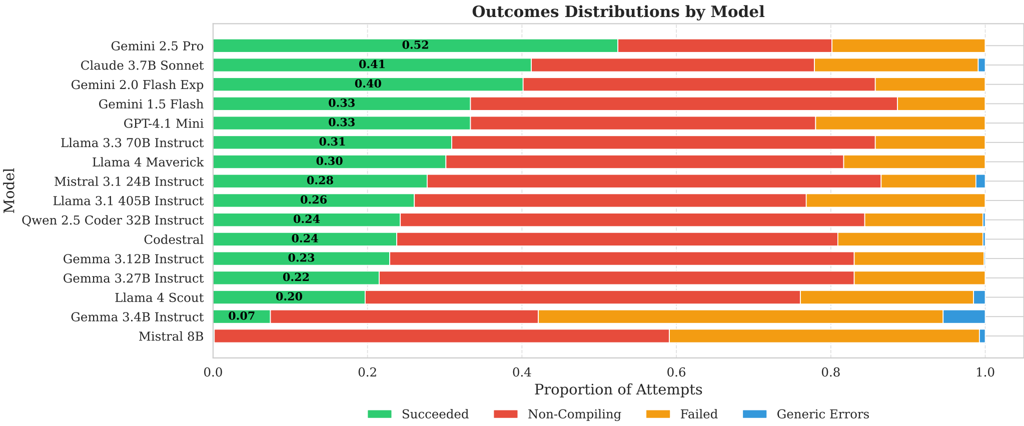


Fig. 7. Distribution of errors across different test cases.

Figure 8 illustrate the impact of the BoK on model performance. The introduction of a BoK leads to a significant performance improvement across all models (as indicated by the rising black average line in the figure). Unexpectedly, the addition of building blocks does not consistently yield further improvements for all models. Some larger models like *Claude 3.7 Sonnet* show minimal gains or even slight decreases when moving from basic knowledge to building blocks. Smaller models like *Llama 4 Maverick* also demonstrate decreased performance when building blocks are introduced. This behavior may be attributed to the dual nature of building blocks: while they provide powerful abstractions, they also introduce additional complexity that some models struggle to effectively leverage. The semantic gap between understanding basic operators and comprehending their composition into higher-level patterns appears to affect different architecture families differently.

The co-design process (cf. Section 4.1) for the BoK with building blocks on average improves the pass@10 metric, with the most significant gains observed in the *Gemini* models, which show an increase of approximately 20% in pass@10 when using building blocks. Such improvements are observed because the BoK is refined based on the co-design process involving the Gemini models (the most successful ones), which effectively tailors the knowledge representation to align with these models' reasoning patterns.

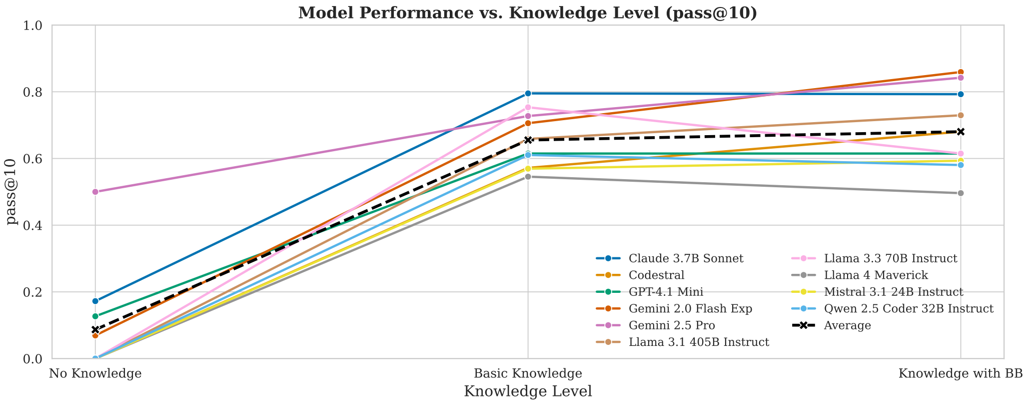


Fig. 8. Analysis of error patterns and knowledge impact on model performance.

5.3.2 Performance Without Body of Knowledge (BoK). As shown in Table 5 (columns "No Knowledge"), the performance in this scenario was generally poor across most models. The pass@10 metric was particularly low, with most models achieving a pass@10 of 0.00, even with big models like *Llama 3.1 405B Instruct*, *Codestral* and *Llama 3.3 70B Instruct*. This indicates that the models struggled to generate even syntactically correct code for the given tasks without any prior knowledge.

The *Gemini 2.5 Pro* model stood out with a pass@10 of 0.50 and compile@10 of 0.77, indicating some latent capability. *Claude 3.7 Sonnet* and *GPT-4.1 Mini* also showed marginal success, with pass@10 values of 0.17 and 0.13, respectively. However, for most models like *Llama 4 Scout* and *Gemini 1.5 Flash*, even achieving syntactically correct code was a challenge, with a compile@10 of 0.00.

The "No Knowledge" heatmap in Figure 9 reveals a stark pattern across all models. The majority of cells are colored dark purple, indicating a pass@10 of 0.00, with only a few scattered successes in the top left corner of the heatmap, where the best-performing models are located. Even though the top-performing model achieves good results on simpler tasks like "Count Down" and "Neighbors Ex Self" (pass@10 of 1.00), it struggles with more complex tasks like "Channel with Obstacles" and "SCR Temp" (pass@10 of 0.00). These scattered successes suggest some latent knowledge of basic ScaFi operators exists within certain models' training data. However, this knowledge appears fragmented and inconsistent, insufficient for reliable code generation without explicit guidance.

5.3.3 Performance with Minimal Body of Knowledge (Basic BoK). Providing a Basic BoK led to a marked improvement in performance for almost all models, as reported in Table 5 (columns "Basic Knowledge"). The average pass@10 across models increased significantly: Claude 3.7 Sonnet (0.80 pass@10), Gemini 2.5 Pro (0.73 pass@10), and Llama 3.3 70B Instruct (0.75 pass@10) emerged as strong performers. Compile rates also saw substantial gains; for example, GPT-4.1 Mini went from a compile@10 of 0.45 (No Knowledge) to 0.99 with the Basic BoK. This indicates that even a concise BoK helps LLMs grasp the syntactic structure of the target language. The "Basic Knowledge" heatmap in Figure 9 shows a general brightening of cells compared with the "No Knowledge" heatmap.

Models like Gemini 2.5 Pro, Claude 3.7 Sonnet, and Gemini 2.0 Flash Exp now successfully tackled a wider range of basic temporal ("Count Down") and spatial ("Neighbors", "Gather IDs") tasks with high pass@10 scores (often 1.00). Performance on simpler spatio-temporal tasks like "Collect Max ID" and "Gradient" also improved. For instance, Gemini 2.5 Pro achieved 1.00 on "Max ID" and "Gradient" (without obstacles).

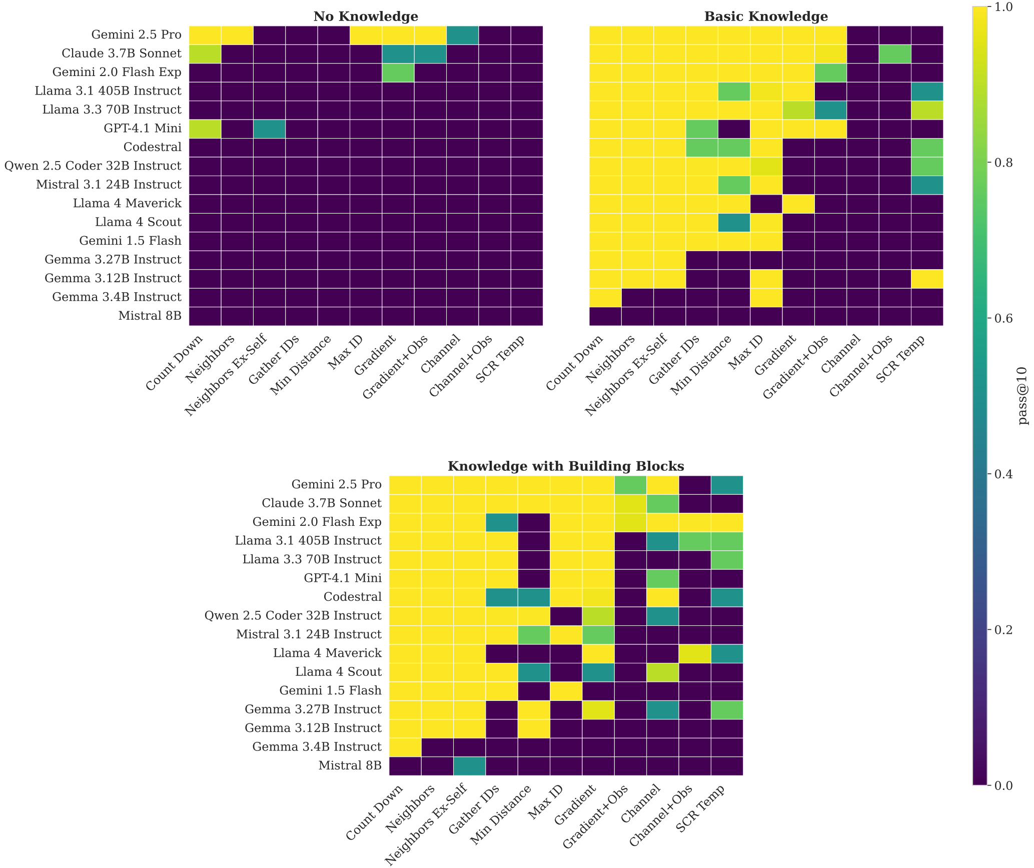


Fig. 9. Comparison of LLMs performance with different knowledge levels.

However, some tasks remained challenging, particularly for more complex spatio-temporal tasks (e.g., “Gradient+Obs”) and tasks implicitly requiring building-block-like patterns (e.g., “Channel”, “SCR Temp”), where pass@10 scores were still low or zero for many models. This suggests that while the Basic BoK helps with fundamental operators, it is not sufficient for guiding LLMs toward composing more elaborate, robust, and self-organizing behaviors. Figure 10, particularly the “Basic Knowledge” plots, further illustrates that while the “Basic” and “Spatio-Temporal” test categories showed improvement, the “BB” (Building Blocks) category likely remained challenging.

5.3.4 Performance with Enhanced Body of Knowledge (BoK with Building Blocks). As detailed in Table 5 (columns “Knowledge+BB”), this enhanced BoK generally led to further improvements, particularly for the more complex tasks. Gemini 2.5 Pro maintained its strong performance with a pass@10 of 0.84 and a compile@10 of 1.00. Gemini 2.0 Flash Exp also showed notable gains, reaching a pass@10 of 0.86. Claude 3.7 Sonnet achieved a pass@10 of 0.79. The compile rates for top models approached or reached 1.00, indicating high syntactic confidence.

Figure 8 shows that for several top-performing models (e.g., Gemini 2.5 Pro, Gemini 2.0 Flash Exp), the pass@10 increased with the “Knowledge with Building Blocks”. The average pass@10 also saw an uptick. The most significant impact of the enhanced BoK is visible in the “Knowledge with Building Blocks” heatmap in Figure 9. Models now demonstrate a markedly improved ability

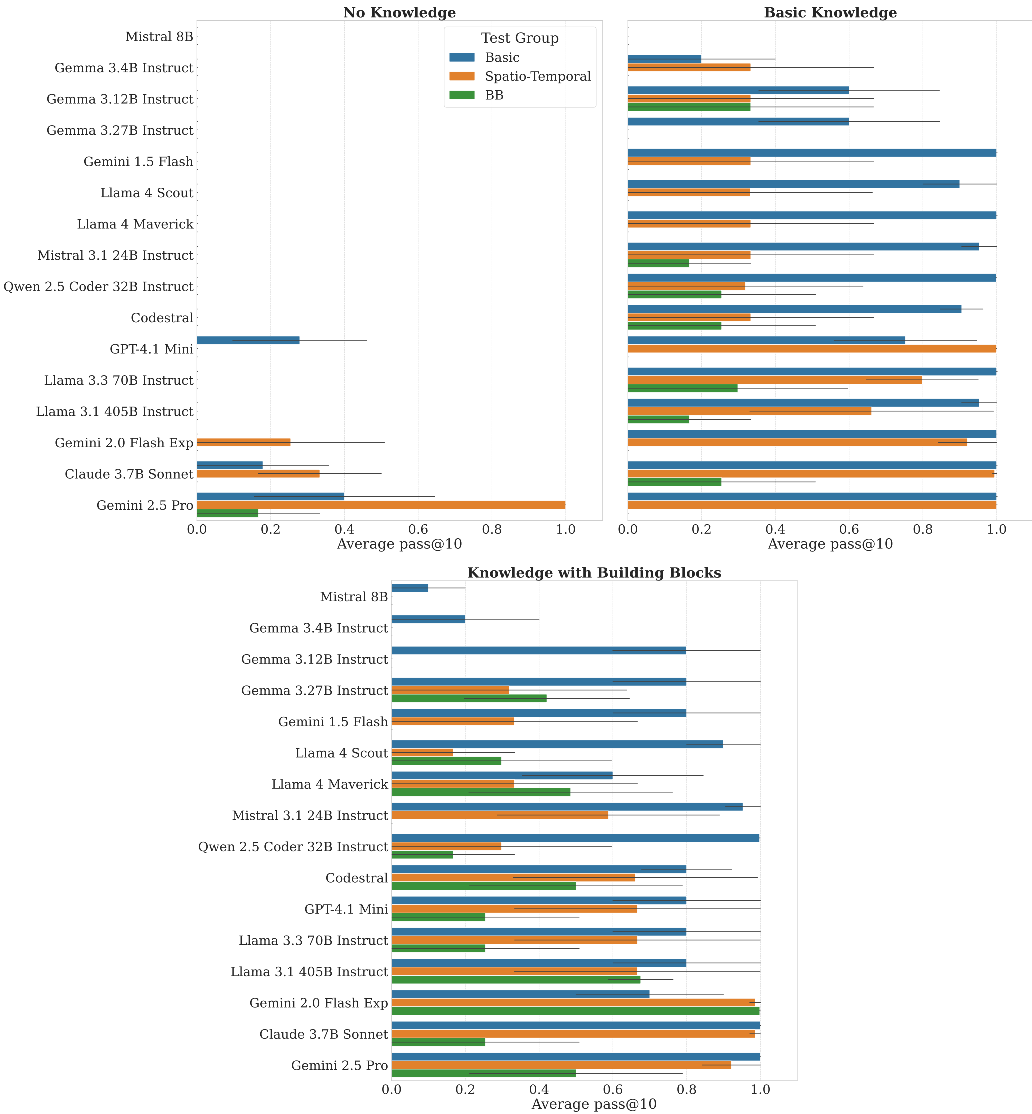


Fig. 10. Comparison of LLMs performance across different test groups.

to solve tasks that were previously unfeasible. For instance, Gemini 2.5 Pro achieves a `pass@10` of 0.76 on “Channel+Obs” and 0.70 on “SCR Temp” (Area-wise Alarm).

However, adding the advanced **Building Blocks** (**BB**) knowledge did not uniformly improve results. As reported in Table 5 and visible in Figures 8 and 9, several models actually saw performance drops:

- **Llama 3.3 70B Instruct:** `pass@10` dropped from 0.75 (Basic BoK) to 0.61 (BoK+BB), and `compile@10` dropped from 0.97 to 0.80.
- **Llama 4 Maverick:** `pass@10` dropped from 0.55 to 0.50, while `compile@10` dropped from 0.90 to 0.80.
- **Claude 3.7 Sonnet:** remained almost unchanged (pass@10 from 0.80 to 0.79).

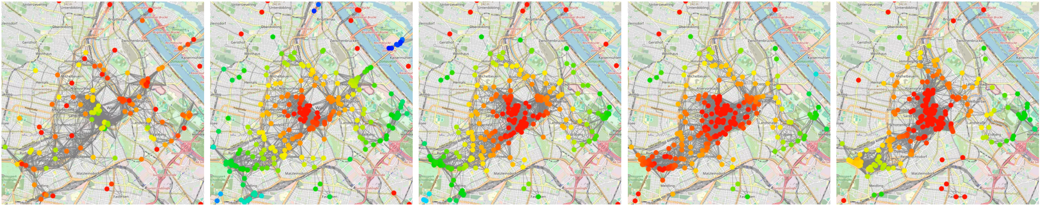


Fig. 11. Visualization of crowd detection algorithm applied to the real-case marathon simulation in Alchemist. Red nodes indicate highly crowded areas, while blue nodes represent areas with lower crowd density. The panels show the temporal evolution from left to right. Crowded zones dynamically shift over time as device density changes across different areas of the marathon route.

```
val DENSITY_THRESHOLD_COUNT = 50
val localNeighboringDeviceCount = foldhoodPlus(0)(_ + _)(nbr(1))
val isLocallyOvercrowded = localNeighboringDeviceCount >= DENSITY_THRESHOLD_COUNT
val distanceToNearestCrowdedArea = G[Double](
  source = isLocallyOvercrowded,
  field  = 0.0,
  acc    = pathDistanceSoFar => pathDistanceSoFar + nbrRange(),
  metric = nbrRange
)
distanceToNearestCrowdedArea
```

Listing 1. Distance to the nearest overcrowded area

Fig. 12. Aggregate computing snippet to compute the distance to the nearest overcrowded area.

This phenomenon suggests three key challenges:

(1) *Increased Reasoning Complexity:* models may over-apply high-level abstractions even when simpler foldhood/nbr patterns suffice, leading to convoluted or incorrect code.
(2) *Cognitive Overload or Interference:* a richer BoK can expand the decision space, making it harder for the LLM to select the optimal construct.
(3) *Misinterpretation of Applicability:* LLMs still struggle to discern when a building block is truly the most parsimonious solution versus composing behavior from core field-calculus operators.

In summary, while self-organizing building blocks can unlock solutions to complex compositional problems for some models (see the uplift in the "BB" category of Figure 10), they can also introduce new errors or degrade performance on tasks the models previously handled well.

Real-World Scenario. Given the crowding detection scenario described in Section 2 as a prompt, our LLM, equipped with a BoK that includes self-organizing building blocks and corrections, generated the aggregate computing code snippet shown in Figure 12. Analyzing the generated code, we can see that it effectively implements the two key components of the crowd detection algorithm. First, it establishes a local detection system using spatial operators (`foldhoodPlus`) to identify areas where device concentration exceeds a predefined threshold. Second, it employs the gradient cast building block (`G`) to propagate information about crowded areas throughout the network, creating a potential field that quantifies proximity to congested zones.

In our Alchemist simulation using the city-marathon dataset, we observed that this implementation successfully fulfilled all three evaluation criteria (see Figure 11): (1) accurate identification of high-density areas through local neighborhood assessment (see the red nodes in the figure), (2) efficient propagation of crowding information through spatial gradients (see the variation of the

color over time), and (3) adaptive response to dynamic topology changes as participants moved throughout the event area (see the evolution of the snapshots).

5.4 Discussion

RQ1: *Are state-of-the-art LLMs capable of generating macro-code without a BoK?* Our experimental results indicate that state-of-the-art LLMs, despite their sophisticated reasoning capabilities, demonstrate significant limitations in generating valid aggregate computing code without a BoK. Quantitatively, the absence of a structured BoK leads to a high percentage of non-compiling code, with even the most advanced LLMs (e.g., `Claude`) achieving only a c@1 of 15%. However, considering the *Calculate Gradient* case, the successful attempts produce code similar to the one provide as follows:

```
rep(Double.PositiveInfinity) { d =>
  mux(sense[Boolean]("source")) { 0.0 } {
    minHoodPlus(nbr(d) + nbrRange())
  }
}
```

This suggests some foundational knowledge of aggregate computing exists within the LLMs' training data, while however, highlight the fact that the limited exposure to examples during pre-training makes it difficult for the models to consistently access and apply this knowledge. These findings demonstrate the necessity of a well-structured BoK to effectively guide LLMs in generating reliable aggregate computing code. The BoK serves as a crucial mechanism to both *activate* and *contextualize* the models' latent domain knowledge.

RQ2: *Are prompt engineering techniques sufficient to convey the abstractions of a macro paradigm?* To evaluate the effectiveness of prompt engineering in conveying macroprogramming abstractions, we analyzed performance across three dimensions: temporal (RQ2.1), spatial (RQ2.2), and spatio-temporal (RQ2.3) abstractions. Our analysis indicates that the effectiveness of prompt engineering varies significantly across these dimensions. For temporal abstractions (RQ2.1), results show high efficacy with p@10 approaching 100% for basic patterns such as counting or state evolution. Instead, spatial abstractions (RQ2.2) demonstrate reasonable effectiveness ($\sim$ 80% p@10), though edge cases reveal limitations. Consider this attempted solution for computing minimum neighbor distance:

```
val distances = foldhoodPlus(List.empty[(Double, ID)])(_ ++ _)(List(nbrRange() ->
    nbr(mid())))
distances.minBy(_._1)._1
```

This implementation, while conceptually sound, fails due to operational details—specifically using `foldhoodPlus` without handling empty list cases. This demonstrates that LLMs can grasp spatial concepts but may miss subtle implementation requirements.

For spatio-temporal abstractions (RQ2.3), prompt engineering exhibits significant limitations, with p@10 dropping to 45% for `Gemini 2.5 Pro` and 25% for smaller models. Consider this flawed gradient implementation (Gemini 2.0 Flash):

```
rep[Option[Double]](if (sense[Boolean]("source")) Some(0.0) else None) { distance =>
  mux(distance.isDefined && !sense[Boolean]("obstacle")) {
    minHood(nbr(distance).map(_ + nbrRange()))
  } { distance }
}.getOrElse(Double.PositiveInfinity)
```

Here, the model fundamentally misunderstands nbr as returning a collection rather than accessing a single neighbor value, demonstrating semantic misconceptions despite syntactic correctness. Another common error involves confusing `branch` and `mux`:

```
rep(Double.MaxValue) { dist =>
  branch(sense[Boolean]("source")) { 0.0 } { minHood(nbr(dist) + nbrRange()) }
}
```

This error reflects a misunderstanding of fundamental semantics—`mux` performs value selection while `branch` creates isolated computational domains.

Model size significantly impacts performance, with larger models generally demonstrating better comprehension of abstractions. The `Gemini 2.5 Pro` and `Claude 3.7 Sonnet` models achieved the highest success rates across all abstraction types. However, even large models struggled with complex spatio-temporal tasks requiring composition of multiple operators.

These quantitative results (see Section 5.3) show that introducing building blocks into the BoK enables LLMs—especially the top-performing models—to solve more complex macroprogramming tasks that require higher-level composition and collective reasoning. While this effect is most pronounced for the Gemini family, the overall trend confirms that richer abstractions can bridge the gap between natural language specifications and executable collective code. However, as previously discussed, this benefit is not uniform across all models: some smaller or less capable models may experience diminished performance, likely due to increased abstraction complexity exceeding their reasoning capacity.

RQ3: *Given RQ2, are these LLMs able to generate novel/unseen solutions to macroprogramming problems using at least one macroprogramming language?* For our third research question, we primarily examine solutions adopting building blocks, where the higher abstraction level provides greater potential for novel approaches. When analyzing the generated code for Gemini 2.5 pro, and we identified several instances where the LLM have created innovative solutions that diverged from conventional aggregate computing patterns taught in the BoK. One particularly interesting example appeared in the *Create Channel* task with obstacles:

```
val obstacleDetected: Boolean = sense[Boolean]("obstacle")
val destination: Boolean = sense[Boolean]("destination")
val source: Boolean = sense[Boolean]("source")
val metric = () => if (obstacleDetected) Double.PositiveInfinity else nbrRange()
val potential = G[Double](source, 0.0, _ + nbrRange(), metric)
C[Double, Boolean](potential, _ || _, destination, false)
```

This has two main novelty points:

(1) The use of the C (collect-cast) operator to create the channel, which represents an innovative approach not explicitly taught in the BoK. Traditionally, channels are created using gradient-based path formation, but the model discovered that collect-cast can elegantly propagate boolean values along the shortest path between source and destination nodes.
(2) The implementation of obstacle avoidance through metric manipulation (setting `Double.PositiveInfinity` for obstacle nodes) rather than using `branch` operators. This approach demonstrates sophisticated spatial reasoning where the model treats obstacles as regions with infinite distance, naturally forcing the gradient (and resulting channel) to route around them.

The most significant finding emerged from the temperature analysis task, which required both spatio-temporal reasoning and composition of multiple behaviors to correctly identify areas with temperatures exceeding a predefined threshold. The model generated the following solution:

```
val isLeader = S(2, nbrRange)
val temperature = sense[Double]("temperature")
val potential = G(isLeader, 0.0, _ + nbrRange(), nbrRange)
val areaTemperature = C[Double, Double](potential, _ + _, temperature, 0.0)
val areaSize = C[Double, Int](potential, _ + _, 1, 0)
val averageTemperature = if (areaSize > 0) areaTemperature / areaSize else 0.0
val alarm = averageTemperature > 30.0
G(isLeader, alarm, (a:Boolean) => a, nbrRange)
```

This solution demonstrates the model's ability to independently discover and implement the **Self-organizing Coordinated Regions** (**SCR**) pattern—a fundamental design pattern in aggregate computing [46]. The variable naming conventions (`isLeader`, `potential`, `areaTemperature`) indicate the model's semantic understanding of core aggregate computing concepts: (1) leader election through the sparse choice operator, (2) potential field generation to define areas of influence, and (3) data aggregation and processing within these defined regions. Without the building blocks provided in the enhanced BoK, the abstraction gap is too high for the models to bridge it, resulting in solutions that either failed to compile or exhibited incorrect behavior. With the introduction of these higher-level abstractions, the models were able to reason effectively about collective spatial behaviors and compose multiple operations to achieve the desired outcome. These results qualitatively demonstrate that LLMs can generate novel, functionally correct solutions to complex macroprogramming problems when provided with appropriate abstractions that reduce the semantic distance between natural language specifications and executable code. The building blocks act as conceptual bridges that enable LLMs to translate high-level spatial and temporal concepts into concrete implementations.

6 Threats to Validity

Several factors could influence the interpretation and generalizability of our findings.

External Validity: Our results are based on a specific set of LLMs and the aggregate computing paradigm implemented in ScaFi. While we tested diverse models (mitigating LLM specificity) and aggregate offers space-time universality [4] (justifying paradigm choice), findings may not directly transfer to untested LLMs or entirely different macroprogramming paradigms. The representativeness of our 11 test cases and the specific design of our BoK also constitute limitations; however, test cases were systematically categorized by complexity, and the BoK's progressive structure and content are transparently provided for scrutiny.

Construct Validity: Operationalizing "understanding" of abstractions (RQ2) through `pass@k` is a practical proxy, which we supplemented with qualitative analysis of generated code to gain deeper insights beyond functional correctness. Similarly, assessing "novelty" (RQ3) can be subjective, but we grounded it by comparing generated solutions to BoK content and known aggregate computing patterns. LLM outputs are sensitive to prompt engineering; we addressed this by using a standardized template and fixed formatting requirements, though inherent LLM sensitivity remains. The metrics `pass@k` and `compile@k` primarily capture functional and syntactic correctness, not necessarily code quality or efficiency, a gap partially bridged by our qualitative review.

Internal Validity: Attributing performance changes solely to BoK content is challenging. While improvements with BoK introduction are clear, disentangling the effects of specific content (e.g., building blocks) from increased context length or example density is difficult. However, instances where adding building blocks *decreased* performance for some models suggest content complexity, not just quantity, is influential. Whether LLMs "learn" from the BoK or use it to better retrieve pre-existing knowledge is an open interpretability challenge in LLM research.

Conclusion Validity and Reliability: LLMs are stochastic; we mitigated this by conducting 20 repetitions per test and using `pass@k` metrics. For API-based LLMs, unannounced model updates could affect long-term reproducibility, though we specified model versions where possible and our evaluation framework is open. The co-design of the BoK and qualitative analyses involve human judgment; this was mitigated by collaborative author input and making artifacts publicly available. There is a potential risk that the BoK was implicitly tuned to the test cases during the co-design phase, which we acknowledge as a limitation, though the aim was to capture fundamental AC concepts.

LLM Trustworthiness and Hallucinations: A significant inherent challenge with LLMs is their propensity to "hallucinate"—generating outputs that are plausible-sounding but factually incorrect, inconsistent, or nonsensical. In the context of code generation, this can manifest as syntactically valid but semantically flawed code, misapplication of operators (e.g., confusing `mux` with `branch` despite BoK entries), or the invention of non-existent API features. Such hallucinations directly threaten the trustworthiness of the generated code and the validity of our evaluation. While our methodology, employing unit tests, a structured BoK, and iterative refinement via the human-in-the-loop co-design process, aims to mitigate these risks by grounding the LLM's output, it cannot entirely eliminate them. The "pass@k" metric itself can be influenced if a hallucinated solution coincidentally passes tests. Future work could explore more robust grounding techniques or verifiable reasoning steps to further enhance the reliability of LLM-generated macroprograms.

Applicability in Real-World or Industrial Environments: While our evaluation demonstrates the potential of LLMs to generate aggregate computing code and its applicability in simulated IoT scenarios, deployment in real-world industrial environments requires additional considerations. Production systems demand robust safety mechanisms, including runtime monitoring, fallback strategies, and dynamic validation to handle unexpected conditions that may not be captured in simulation. Although aggregate computing may also support runtime monitoring [5] through its field-based abstractions, enabling human operators to define safety checks that halt execution when necessary, our current approach focuses primarily on code generation and correctness validation. Future work should explore integration frameworks that ensure generated programs meet industrial reliability standards and can be safely deployed in mission-critical IoT systems.

7 Conclusion

In this article, we have addressed the challenge of programming IoT systems by proposing a language-based approach that leverages LLMs to generate aggregate computing code. Our work is motivated by the recent convergence of two significant technological trends: the emergence of sophisticated LLMs with unprecedented reasoning capabilities, and a paradigm shift toward collective system programming rather than individual device programming.

Our co-design approach demonstrates how LLMs can effectively generate code for IoT systems using aggregate computing, a macroprogramming paradigm that abstracts the complexity of distributed systems. This approach offers several significant advantages for IoT development. First, it abstracts complexity by focusing on collective behavior rather than individual devices, enabling developers to specify system-wide properties in natural language. Second, the generated code inherits the self-stabilization, resilience to transient faults, and scalability properties of the aggregate computing paradigm. Finally, our approach lowers the expertise barrier by making macroprogramming more accessible, thereby bridging the gap between high-level system requirements and executable distributed code.

Our experimental evaluation reveals insights into LLMs' capabilities for macroprogramming. The models demonstrate a strong understanding of basic temporal and spatial operators, whereas

spatio-temporal operators and higher-order composition pose greater challenges. The results demonstrate that the introduction of building blocks through the co-design process yields mixed but generally positive outcomes across different model families. While top-performing models like `Gemini 2.5 Pro` and `Gemini 2.0 Flash Exp` show substantial improvements (approximately 15% increase in pass@10), some models experience performance degradation when faced with increased abstraction complexity. Most notably, our findings indicate that LLMs can generate novel solutions that diverge from conventional patterns taught in the BoK, such as innovative use of collect-cast operators for channel creation and independent discovery of the SCR pattern. These results suggest that appropriate abstractions serve as conceptual bridges, enabling effective translation from natural language specifications to executable collective code.

Future research directions include: investigating automated prompt optimization techniques such as textgrad [62] to systematically enhance knowledge transfer between human experts and LLMs; exploring synergistic approaches that combine in-context learning with fine-tuning methodologies to improve performance in IoT code generation; developing domain-specific prompt optimizations tailored to various IoT application domains (e.g., smart cities, precision agriculture, industrial IoT); explore further validation checks to mitigate the risk of hallucinations in generated code before deployment of the generated code in real-world scenarios; and leveraging a LLM as a judge framework to automatically evaluate the correctness of generated code against a set of predefined criteria, such as functional correctness, performance, and resource efficiency. Through these advancements, we aim at furthering bridge the gap between natural language requirements and robust distributed systems implementations, making macroprogramming accessible to a broader audience of developers and domain experts.

References

[1] Gregory D. Abowd. 2016. Beyond weiser: From ubiquitous to collective computing. 1 (2016), 17–23. DOI: https://doi.org/10.1109/MC.2016.22

[2] Gianluca Aguzzi. 2024. *A Language-Based Software Engineering Approach for Cyber-Physical Swarms*. Ph.D. Dissertation. University of Bologna, Italy. Retrieved from http://amsdottorato.unibo.it/11386/

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. 4 (2010), 50–58. DOI: https://doi.org/10.1145/1721654.1721672

[4] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. 2018. Space-time universality of field calculus. In *Coordination Models and Languages—20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings (Lecture Notes in Computer Science)*. Giovanna Di Marzo Serugendo and Michele Loreti (Eds.), Vol. 10852. Springer, 1–20. DOI: https://doi.org/10.1007/978-3-319-92408-3_1

[5] Giorgio Audrito, Ferruccio Damiani, Volker Stolz, and Mirko Viroli. 2018. On distributed runtime verification by aggregate computing. In *Proceedings of the Second Workshop on Verification of Objects at RunTime EXecution, VORTEX@ECOOP/ISSTA 2018, Amsterdam, Netherlands, 17th July 2018 (EPTCS)*, Davide Ancona and Gordon Pace (Eds.), Vol. 302. 47–61. DOI: https://doi.org/10.4204/EPTCS.302.4

[6] Giorgio Audrito and Gianluca Torta. 2024. FCPP to aggregate them all. (2024), 103026. DOI: https://doi.org/10.1016/J.SCICO.2023.103026

[7] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. 2019. A higher-order calculus of computational fields. 1 (2019), 5:1–5:55. DOI: https://doi.org/10.1145/3285956

[8] Andrea Azzara, Daniele Alessandrelli, Matteo Petracca, and Paolo Pagano. 2014. Demonstration abstract: PyoT, a macroprogramming framework for the IoT. IEEE/ACM, 315–316. Retrieved from http://dl.acm.org/citation.cfm?id=2602392

[9] Debasis Bandyopadhyay and Jaydip Sen. 2011. Internet of things: Applications and challenges in technology and standardization. 1 (2011), 49–69. DOI: https://doi.org/10.1007/S11277-011-0288-5

[10] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2012. Organizing the aggregate: Languages for spatial computing. (2012). arXiv:1202.5509. Retrieved from https://arxiv.org/abs/1202.5509

[11] Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate programming for the internet of things. 9 (2015), 22–30. DOI: https://doi.org/10.1109/MC.2015.261

[12] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2017. Self-adaptation to device distribution in the internet of things. 3 (2017), 12:1–12:29. DOI: https://doi.org/10.1145/3105758

[13] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. 2013. Swarm robotics: A review from the swarm engineering perspective. 1 (2013), 1–41. DOI: https://doi.org/10.1007/S11721-012-0075-2

[14] Roberto Casadei. 2023. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. 13s (2023), 275:1–275:37. DOI: https://doi.org/10.1145/3579353

[15] Roberto Casadei, Gianluca Aguzzi, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, Giordano Scarso, Gianluca Torta, and Mirko Viroli. 2025. Software engineering for collective cyber-physical ecosystems. (Jan. 2025). DOI: https://doi.org/10.1145/3712004 Just Accepted.

[16] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. 2022. ScaFi: A scala DSL and toolkit for aggregate programming. (2022), 101248. DOI: https://doi.org/10.1016/J.SOFTX.2022.101248

[17] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2023. Unleashing the potential of prompt engineering in Large Language Models: A comprehensive review. (2023). DOI: https://doi.org/10.48550/ARXIV.2310.14735 arXiv:2310.14735

[18] Hongwei Cui, Yuyang Du, Qun Yang, Yulin Shao, and Soung Chang Liew. 2023. LLMind: Orchestrating AI and IoT with LLMs for complex task execution. (2023). DOI: https://doi.org/10.48550/ARXIV.2312.09007 arXiv:2312.09007

[19] Jasenka Dizdarevic, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. 2019. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. 6 (2019), 116:1–116:29. DOI: https://doi.org/10.1145/3292674

[20] Ran Eshel and Yael Moses. 2008. Homography based multiple camera detection and tracking of people in a dense crowd. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*. IEEE Computer Society. DOI: https://doi.org/10.1109/CVPR.2008.4587539

[21] Ran Eshel and Yael Moses. 2010. Tracking in a dense crowd using multiple cameras. 1 (2010), 129–143. DOI: https://doi.org/10.1007/S11263-009-0307-0

[22] Nicolas Farabegoli and Gianluca Aguzzi. 2025. nicolasfara/experiments-2025-acm-iot-ac-llm: v1.4.0. (May 2025). DOI: https://doi.org/10.5281/zenodo.15516944

[23] Nicolas Farabegoli, Danilo Pianini, Roberto Casadei, and Mirko Viroli. 2024. Scalability through pulverisation: Declarative deployment reconfiguration at runtime. (2024), 545–558. DOI: https://doi.org/10.1016/J.FUTURE.2024.07.042

[24] Nicolas Farabegoli, Mirko Viroli, and Roberto Casadei. 2024. Flexible self-organisation for the cloud-edge continuum: A macro-programming Approach. IEEE, 21–30. DOI: https://doi.org/10.1109/ACSOS61780.2024.00020

[25] Alois Ferscha. 2015. Collective adaptive systems. ACM, 893–895. DOI: https://doi.org/10.1145/2800835.2809508

[26] Cristian González García, Daniel Meana-Llorián, Vicente García-Díaz, Andrés Camilo Jiménez, and John Petearson Anzola. 2020. Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering. (2020), 141872–141894. DOI: https://doi.org/10.1109/ACCESS.2020.3012503

[27] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using *Kairos (Lecture Notes in Computer Science)*. Springer, 126–140. DOI: https://doi.org/10.1007/11502593_12

[28] Fatma-Zohra Hannou, Maxime Lefrançois, Pierre Jouvelot, Victor Charpenay, and Antoine Zimmermann. 2024. A Survey on IoT Programming Platforms: A Business-Domain Experts Perspective. 4, Article 80 (Dec. 2024), 37 pages. DOI: https://doi.org/10.1145/3699954

[29] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: A language and code generation framework for heterogeneous targets. ACM, 125–135. Retrieved from http://dl.acm.org/citation.cfm?id=2976812

[30] Ahmed E. Hassan, Gustavo Ansaldi Oliva, Dayi Lin, Boyuan Chen, and Zhen Ming Jiang. 2024. Towards AI-native software engineering (SE 3.0): A vision and a challenge roadmap. (2024). DOI: https://doi.org/10.48550/ARXIV.2410.06107 arXiv:2410.06107

[31] Junda He, Christoph Treude, and David Lo. 2024. LLM-based multi-agent systems for software engineering: Vision and the road ahead. (2024). DOI: https://doi.org/10.48550/ARXIV.2404.04834 arXiv:2404.04834

[32] 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. OpenReview.net. Retrieved from https://openreview.net/forum?id=VtmBAGCN7o

[33] Sinclair Hudson, Sophia Jit, Boyue Caroline Hu, and Marsha Chechik. 2024. A software engineering perspective on testing large language models: Research, practice, tools and benchmarks. (2024). DOI: https://doi.org/10.48550/ARXIV.2406.08216 arXiv:2406.08216

[34] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. (2024). DOI: https://doi.org/10.48550/ARXIV.2406.00515 arXiv:2406.00515

[35] Carlos Kamienski, Juha-Pekka Soininen, Markus Taumberger, Ramide Dantas, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and André Torre Neto. 2019. Smart water management platform: IoT-based precision irrigation for agriculture. 2 (2019), 276. DOI: https://doi.org/10.3390/S19020276

[36] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[37] Ibrahim Kok, Orhan Demirci, and Suat Ozdemir. 2024. When IoT meet LLMs: Applications and challenges. (2024). DOI: https://doi.org/10.48550/ARXIV.2411.17722 arXiv:2411.17722

[38] Ajay Krishna, Michel Le Pallec, Alejandro Martinez, Radu Mateescu, and Gwen Salaün. 2020. MOZART: Design and deployment of advanced IoT applications. ACM / IW3C2, 163–166. DOI: https://doi.org/10.1145/3366424.3383532

[39] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based pseudocode to code. 11883–11894. Retrieved from https://proceedings.neurips.cc/paper/2019/hash/7298332f04ac004a0ca44cc69ecf6f6b-Abstract.html

[40] Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. 2003. State-centric programming for sensor-actuator network systems. 4 (2003), 50–62. DOI: https://doi.org/10.1109/MPRV.2003.1251169

[41] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David E. Culler. 2002. Supporting aggregate queries over Ad-Hoc wireless sensor networks. IEEE Computer Society, 49–58. DOI: https://doi.org/10.1109/MCSA.2002.1017485

[42] Matteo Magnini, Gianluca Aguzzi, and Sara Montagna. 2025. Open-source small language models for personal medical assistant chatbots. (2025), 100197. DOI: https://doi.org/10.1016/j.ibmed.2024.100197

[43] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. 2004. Co-Fields: A physically inspired approach to motion coordination. 2 (2004), 52–61. DOI: https://doi.org/10.1109/MPRV.2004.1316820

[44] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. arXiv:2307.06435. Retrieved from https://arxiv.org/abs/2307.06435

[45] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. ACM, 489–498. DOI: https://doi.org/10.1145/1236360.1236422

[46] Danilo Pianini, Roberto Casadei, Mirko Viroli, and Antonio Natali. 2021. Partitioned integration and coordination via the self-organising coordination regions pattern. (2021), 44–68. DOI: https://doi.org/10.1016/J.FUTURE.2020.07.032

[47] Danilo Pianini, Sara Montagna, and Mirko Viroli. 2013. Chemical-oriented simulation of computational systems with ALCHEMIST. 3 (2013), 202–215. DOI: https://doi.org/10.1057/JOS.2012.27

[48] Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: practical aggregate programming. ACM, 1846–1853. DOI: https://doi.org/10.1145/2695664.2695913

[49] Farhad Pourpanah, Moloud Abdar, Yuxuan Luo, Xinlei Zhou, Ran Wang, Chee Peng Lim, Xi-Zhao Wang, and Q. M. Jonathan Wu. 2023. A Review of Generalized Zero-Shot Learning Methods. 4 (2023), 4051–4070. DOI: https://doi.org/10.1109/TPAMI.2022.3191696

[50] Yosef Saputra, Jie Hua, Nathaniel Wendt, Christine Julien, and Gruia-Catalin Roman. 2019. WARBLE: Programming abstractions for personalizing interactions in the internet of things. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 128–139. DOI: https://doi.org/10.1109/MOBILESoft.2019.00026

[51] Mahadev Satyanarayanan. 2001. Pervasive computing: vision and challenges. 4 (2001), 10–17. DOI: https://doi.org/10.1109/98.943998

[52] Mahadev Satyanarayanan. 2017. The emergence of edge computing. 1 (2017), 30–39. DOI: https://doi.org/10.1109/MC.2017.9

[53] Ognjen Scekic, Tommaso Schiavinotto, Svetoslav Videnov, Michael Rovatsos, Hong Linh Truong, Daniele Miorandi, and Schahram Dustdar. 2020. A Programming Model for Hybrid Collaborative Adaptive Systems. 1 (2020), 6–19. DOI: https://doi.org/10.1109/TETC.2017.2702578

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. 5998–6008. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[55] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2018. Engineering resilient collective adaptive systems by self-stabilisation. 2 (2018), 16:1–16:28. DOI: https://doi.org/10.1145/3177774

[56] Mirko Viroli, Roberto Casadei, and Danilo Pianini. 2016. On execution platforms for large-scale aggregate computing. ACM, 1321–1326. DOI: https://doi.org/10.1145/2968219.2979129

[57] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2021. Generalizing from a few examples: A survey on few-shot learning. 3 (2021), 63:1–63:34. DOI: https://doi.org/10.1145/3386252

[58] Mark D. Weiser. 1999. The computer for the 21st century. 3 (1999), 3–11. DOI: https://doi.org/10.1145/329124.329126

[59] Matt Welsh and Geoffrey Mainland. 2004. Programming sensor networks using abstract regions. USENIX, 29–42. Retrieved from http://www.usenix.org/events/nsdi04/tech/welsh.html

[60] Bin Xiao, Burak Kantarci, Jiawen Kang, Dusit Niyato, and Mohsen Guizani. 2025. Efficient prompting for LLM-based generative internet of things. 1 (2025), 778–791. DOI: https://doi.org/10.1109/JIOT.2024.3470210

[61] Burak Yetistiren, Isik Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. (2023). DOI: https://doi.org/10.48550/ARXIV.2304.10778 arXiv:2304.10778

[62] Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. TextGrad: Automatic "Differentiation" via text. (2024). DOI: https://doi.org/10.48550/ARXIV.2406.07496 arXiv:2406.07496

[63] Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. De Angelis, Giovanna Di Marzo Serugendo, Simon A. Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, et al. 2015. Developing pervasive multi-agent systems with nature-inspired coordination. (2015), 236–252. DOI: https://doi.org/10.1016/J.PMCJ.2014.12.002

[64] Ningze Zhong, Yi Wang, Rui Xiong, Yingyue Zheng, Yang Li, Mingjun Ouyang, Dan Shen, and Xiangwei Zhu. 2024. CASIT: Collective intelligent agent system for internet of things. 11 (2024), 19646–19656. DOI: https://doi.org/10.1109/JIOT.2024.3366906

[65] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. (2024). DOI: https://doi.org/10.48550/ARXIV.2404.02525 arXiv:2404.02525

[66] Mingyu Zong, Arvin Hekmati, Michael Guastalla, Yiyi Li, and Bhaskar Krishnamachari. 2025. Integrating large language models with internet of things: Applications. 1 (2025), 2. DOI: https://doi.org/10.1007/S43926-024-00083-4

Received 4 March 2025; revised 30 May 2025; accepted 7 July 2025