Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2026-01
April 29, 2026

Department of Distributed and Dependable Systems D3S

Component-based Development for the Era of AI Coding Agents

Tomáš Bureš, František Plášil, Petr Hnětynka, Michal Töpfer

Abstract: The AI coding agents have recently received significant industry attention, popularity, and adoption, prompting enormous efforts from their providers—tech giants competing with one another. However, the underlying LLMs have difficulty considering broader context and implicit dependencies. In larger systems, this causes AI coding agents to struggle with issues such as uncertainty about code functionality, repeated design runs, and regression-prone changes. In this position paper, we claim that a promising approach to partitioning and simplifying the context for the AI coding agent is to adopt CBSE concepts (yet reshaped for use by LLMs). To benefit from the key CBSE notions (encapsulation, lifecycle, and controlled interaction only via explicitly defined interfaces), this must be done under specific rules and conditions, which are technically expressed as templates in the skills provided to the AI coding agent. This claim is supported by an example that demonstrates this approach in the context of AI-based CBSE development for ESP32 embedded systems.

Component-based Development for the Era of AI Coding Agents

Tomáš Bureš, František Plášil, Petr Hnětynka, Michal Töpfer

Charles University, Czech Republic
{tomas.bures, frantisek.plasil, petr.hnetynka, michal.topfer}@matfyz.cuni.cz

Abstract. The AI coding agents have recently received significant industry attention, popularity, and adoption, prompting enormous efforts from their providers—tech giants competing with one another. However, the underlying LLMs have difficulty considering broader context and implicit dependencies. In larger systems, this causes AI coding agents to struggle with issues such as uncertainty about code functionality, repeated design runs, and regression-prone changes. In this position paper, we claim that a promising approach to partitioning and simplifying the context for the AI coding agent is to adopt CBSE concepts (yet reshaped for use by LLMs). To benefit from the key CBSE notions (encapsulation, lifecycle, and controlled interaction only via explicitly defined interfaces), this must be done under specific rules and conditions, which are technically expressed as templates in the skills provided to the AI coding agent. This claim is supported by an example that demonstrates this approach in the context of AI-based CBSE development for ESP32 embedded systems.

1 Introduction

The groundbreaking progress of LLMs over the past year has given rise to AI coding agents (AI agents later on), e.g., [4, 10], and has had a radically transformative effect on software development. The AI coding agent is generally understood as an LLM equipped with tools that can autonomously carry out tasks like implementation, test construction, build and test execution, consultation of documentation and the web to understand frameworks and tools. This development style is increasingly referred to as *vibe-coding*: the human specifies intent in natural language, steers the process through high-level feedback, avoiding working with source code [13].

In 2025 and early 2026, AI agents moved from experimental developer tools to a major industrial trend adopted by large software development companies. Public statements from large technology companies report that AI agents mediated implementation becomes a mainstream enterprise practice. The speed of this shift is such that so far it has been documented predominantly in gray literature rather than established academic sources [2, 3, 15].

Although improving almost every month, AI agents still work best in greenfield development, since when dealing with an existing codebase featuring implicit

system contexts and dependencies, their reliability falters. In such a setting, the AI agent is forced to reconstruct the design intent from code that was never meant to be an authoritative representation of that intent. The result is brittle implementation, hidden dependencies, architecture drift, and repeated rediscovery of decisions that should have been explicit from the start. Employing AI agents also transforms the role of software architecture: it becomes the primary control surface through which humans express intent and constrain the design context for the AI agent. This gives software architecture renewed practical importance and creates a major opportunity for the software architecture community. We argue that the need to constrain the design context cannot be addressed simply by adding more prompt rules or by decomposing code into smaller modules. The deeper issue is finding the abstraction at which humans and AI agents must cooperate [14, 9]. In this vein, source code is not an appropriate abstraction since it entangles the design intent with the details of the platform and history.

Software components are a particularly suitable alternative because they have traditionally provided explicit boundaries and served as the primary unit of architectural decomposition. A structured component specification can capture responsibility, lifecycle, configuration, and allowed interactions in a form that is both architecturally meaningful to humans and operationally useful to AI agents. In this view, such a specification becomes the single source of truth between the human and the AI agent, while the code becomes a derived artifact.

Such an approach is fundamentally model-driven. The component specification should define abstractions suitable both for AI-assisted software construction and to reflect the concrete domain and platform of the desired applications. Thus, in the modeling perspective, the transformation path runs from platform-independent component concepts to platform-dependent concepts, rules, and concerns; further on it runs from platform-specific component specifications to source code through AI agent automated transformation. For example, for the ESP-IDF embedded system platform [5], the concerns that a component specification must capture include lifecycle transitions, callbacks, tasks, prerequisites, connector types, and test surfaces.

In this position paper, we propose the COMPAID (COMPonent-based AI-assisted Development) method. Since we argue that the abstractions of the underlying component model need to be grounded in a particular domain and platform, we illustrate the COMPAID on its instance for the ESP-IDF platform on ESP32 chips. We pursue three goals: (1) define the COMPAID component model; (2) identify the key abstractions that this component model must provide when grounded in the ESP-IDF platform; and (3) report early experimental evidence from employing COMPAID in the development of an ESP-IDF application.

2 Background and Related Work

Current AI (coding) agents are powerful tools—they can inspect repositories, edit files, compile and test systems, consult documentation and the web, diagnose

failures, and iteratively repair their own output [13]. This makes them useful in substantial parts of the software development process.

AI agents are often provided with *skills*, reusable instruction packages that encode rules, workflows, templates, and domain guidance for recurring classes of tasks [11]. More broadly, they can be embedded in agentic workflows or orchestrations, where multiple steps, tools, or specialized AI agents cooperate in a structured process [12, 7, 6]

However, the predominant focus of current skills and agentic workflows remains code-centric [10, 8]. They guide the AI agent in inspecting, modifying, testing, debugging, and regenerating code fragments. What they typically do not provide is an explicit concept of the encapsulated software component together with the abstractions architecturally capturing the relevant properties of a particular domain and platform [14, 9]. As a consequence, the system intent often remains distributed across prompts, workflow steps, tool output, and code artifacts.

The COMPAID differs in two ways. First, it places a platform-grounded component specification at the center of the development process. Second, it treats this specification as the only source of truth. It defines the intended components, their boundaries, lifecycles, configurations, and allowed interactions before any code exists. The remaining design process is treated as an AI-based model transformation from this specification to code, tests, and application wiring.

We focus on the ESP-IDF [5] platform for embedded devices based on ESP32 chips, highly popular for small IoT systems and rapid prototyping. ESP-IDF is the official software development framework for these devices, thus being a relevant platform for AI-assisted embedded software development.

ESP-IDF already uses the notion of component, but it is primarily a build, packaging, distribution, and dependency-management unit rather than the component abstraction needed for COMPAID. While an ESP-IDF component can declare dependencies and how it participates in the build, it does not define an architectural contract at the code level, such as task ownership, callback interfaces, synchronization and execution context, or lifecycle behavior. Consequently, the ESP-IDF notion of component is useful for compilation and reuse, but for COMPAID purposes, it has to be substantially enhanced.

3 COMPAID method

The proposed COMPAID method is summarized in Fig. 1 From a modeling perspective, platform-independent component concepts are grounded into platform-dependent component concepts to define the COMPAID component model. At the project level, a concrete project (application) is transformed from the user's intent, over component specification (employing the component model) into application code. This transformation is done in phases 1-5 driven by the procedure rules (*skills*) executed by the agents associated with the phases.

In this sense, component specification and skills play complementary roles. The former captures the intended components, their responsibilities, desired

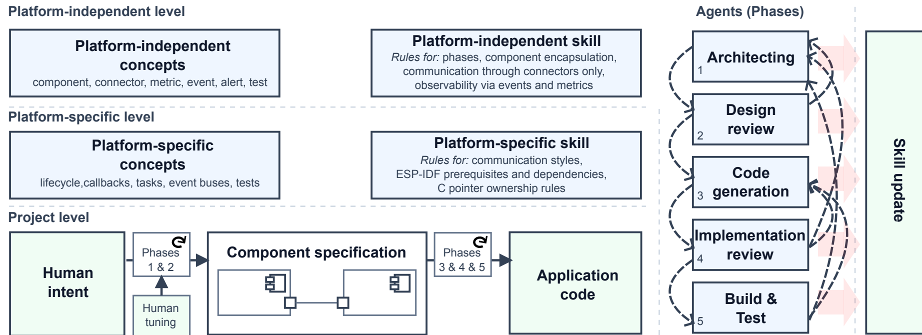

Fig. 1: COMPAID method—overview

interactions, lifecycle, configuration, and test intent. The latter captures the mapping of the component model concepts to platform-specific implementation constructs such as lifecycle functions, callbacks, tasks, event buses, tests, and the review criteria. This is embodied in the collection of rules that govern the process of transforming user intent into valid component specifications and executable artifacts/code. The skills thus operationalize the COMPAID method, making the transformation process repeatable across sessions and across projects.

The project transformation process is realized as a controlled multi-agent activity, organized into five phases with explicit iterative handovers, each of the phases being executed by a dedicated agent:

In the (1) *architecting phase*, the component specification is created. In this task, the user is supported by the architecture agent. The component specification is derived from the (human) intent (Fig. 1), potentially revised by clarification prompts, and manually tuned by the user. In the (2) *design-review phase*, the component specification is audited by the design-review agent with respect to the rules specified by the skills. The component specification and design review are iterated until a consistent design is reached (typically measured by covering the features required for a project milestone). Then, with the explicit approval of the user, the transformation process proceeds to the (3) *code-generation* phase.

Here, the generation agent derives code and tests from the component specification. In the (4) *implementation-review* phase, the generated artifacts are checked by the code-review agent against the transformation process rules defined by the skills. In the (5) *build-and-test* phase, the resulting application is built and tested. Across these phases, several invariants are preserved: the component specification remains the single source of truth, no undeclared inter-component communication is allowed, so that every generated artifact must be traceable back to the component specification. If a problem caused by an inconsistent component specification is discovered, it is reported back to the architecting phase, and the transformation process starts over here.

An important aspect of the transformation process is that corrections/improvements occur at two levels: (i) Within a project, a change is expressed by updating the component specification. (ii) Across projects, the skills are updated when a transformation fails due to a missing rule. This distinction allows the

COMPAID method to evolve without mixing application knowledge (component specification) with transformation process knowledge (skills).

4 ESP-IDF-specific Component Abstractions

The COMPAID method (Section 3) becomes operational only after the component model is specialized to the target platform. For ESP-IDF, the decisive step is to make explicit the operational properties that are architecturally important but likely to be reconstructed incorrectly by an AI agent from code and framework APIs alone. In this setting, correctness depends not only on which functionality a component provides but also on which execution context runs a callback, which shared platform resources must already exist, how data is retained across calls, and which observations will later be used to validate behavior.

In our prototype (Section 5), these abstractions are represented in YAML, complemented by natural-language constraints. Listing 1 shows an example of a component specification (generated from human intent, manually tuned, and structured according to a template in a skill), and the remainder of this section highlights important abstraction concerns. The full semantics of the specification language is part of the AI agent skill and is available in the companion package [1].

A particularly important concern in embedded software is the component lifecycle because failures often stem not from nominal behavior but from hidden assumptions about startup ordering, retained resources, and the state that persists across calls. The lifecycle is thus explicitly mentioned, for example, for state initialization and validity (see line 14) and for startup ordering (18).

The second concern is specifying execution contexts for callbacks and interaction styles for connectors. ESP-IDF systems commonly combine `esp_timer` callbacks, event-loop callbacks, HTTP request handlers, and component-owned FreeRTOS tasks [5]. These contexts are not interchangeable: a long-running timer callback delays other timer callbacks, an event-loop callback delays delivery of other events, etc. For this reason, the callback `description` must specify the execution context and the blocking consequences (22). To make the communication topology across component boundaries explicit and reduce the risk that the code-generation agent will silently introduce ad hoc shared-state or call-path dependencies, the interaction style is explicitly defined for connectors (31, 39). The relevant interaction styles include synchronous calls, event publication and subscription, queue-based message passing, and task notifications.

Another concern that needs to be addressed is the ownership and lifetime of the exchanged data. In C and low-level embedded frameworks, the same interface may be safe or unsafe depending on whether the callee copies a value, borrows a referenced object only for the duration of a call, or retains that reference beyond return. This issue is not limited to raw pointers; it also arises in compound structures containing embedded pointers, callback functions, or handles. Retention and lifetime semantics are therefore explicitly specified in the `description` of the internal state and connectors (40). Otherwise, the agent is

forced to infer memory ownership from framework conventions and surrounding code, which is precisely where regeneration becomes brittle.

Lastly, if the specification is to remain the single source of truth, it must define not only the intended behavior, but also the evidence by which that behavior will later be validated: trace events (43), long-lived observable metrics (50), and, where appropriate, alerts over them (53). It should also separate the component-level test intent (61) from the application-level acceptance intent (68). This distinction matters in a regeneration-oriented process because code, tests, and later review all need to remain accountable to the same explicit contract.

```
 1 component:
 2   id: heartbeat_status
 3   description: Provides a web-page handler which shows the status of the heartbeat component...
 4   espidf_dependencies: [ esp_event, esp_http_server, log ]
 5   tasks: [] # Real-time tasks of the component. This component does not have any.
 6   configuration:
 7     - id: HEARTBEAT_STATUS_URI
 8       type: string
 9       default: /
10       description: URI path served by this component.
11   initialization:
12     stages:
13       - id: setup
14         description: Subscribe to HEARTBEAT_EVENT_TICK on the application event bus. Register the
              HTTP GET handler with http_server via http_server_register_uri.
15         depends_on:
16           - component: heartbeat
17             stage: setup
18             reason: heartbeat must have registered its event base before subscribing.
19   callbacks:
20     - id: heartbeat_tick_callback
21       description:
22         Handles HEARTBEAT_EVENT_TICK from the application event bus. Reads timestamp_us from the
              event payload. Increments ticks_received...  Runs in the ESP-IDF event task context. Blocks the
              default event loop for its duration.
23   internal_state:
24     variables:
25       - id: ticks_received
26         type: uint32_t
27         initial: 0
28         description: Count of HEARTBEAT_EVENT_TICK events received since init.
29   connectors:
30     - id: heartbeat_tick_sub
31       style: event_pubsub
32       description: Receives heartbeat tick notifications to update cached state.
33       bus: application_message_bus
34       role: subscriber
35       event: ...
36       handler: heartbeat_tick_callback
37       payload: ...
38     - id: register_uri_handler
39       style: call
40       description: Calls http_server_register_uri during initialization to register the HTTP GET handler.
            Passes the 'uri_config' internal state by reference because http_server retains that reference until
            unregister or deinit.
41       role: caller
42       ...
43   events:
44     - id: HEARTBEAT_STATUS_TRACE_TICK_RECEIVED
45       description: A heartbeat tick was received and cached state updated.
46       fields:
47         - id: ticks_received
48           type: uint32_t
49           documentation: Updated tick counter.
50   metrics:
51     - id: heartbeat_status_requests_total
52       documentation: Total number of HTTP GET requests served since boot.
53   alerts:
```

```
54     - id: heartbeat_status_no_ticks
55       description: No heartbeat tick has been emitted for more than 2000 ms.
56       severity: error
57       trigger:
58         type: event_absence
59         event: HEARTBEAT_STATUS_TRACE_TICK_RECEIVED
60         window_ms: 3000
61   component_test_plan: # test the component lifecycle and connector behavior in isolation
62   - id: page
63     path: test_apps/page
64     description: Validates the HTTP GET handler output.
65     tests:
66       - HTTP 200 response contains tick count, last interval, and time since last tick
67       - response has Content-Type text/html; charset=utf-8
68   application_test_plan: # test externally visible system behavior once components are composed
69   - id: heartbeat_status_page_accessible
70     doc: Status page is available (returns 200) after a 10 seconds delay needed for WiFi to connect.
```

Listing 1: Abridged excerpts from ESP-IDF component specifications used by the prototype.

5 Experiments, Lessons Learned and Future Work

This position paper presented a specification-first, component-based approach to AI-assisted implementation grounded in ESP-IDF-specific abstractions. We have also created an initial implementation of the COMPAID method, available in the companion package [1]. This implementation realizes the process described in Section 3 through dedicated agents and includes the ESP-IDF skill (referred to in Fig. 1 as a platform-specific rule).

So far, we have experimented with this implementation on an ESP-IDF application composed of four components (also included in the companion package). These initial experiments suggest that the approach is technically feasible and yields a clearer structure and greater understandability than generating the same application in a traditional vibe-coding workflow that centers directly on source code. Also, the architecting agent proved to be quite capable of suggesting a sound structuring of the responsibilities to the components. Nevertheless, the evidence remains preliminary, and a comprehensive evaluation of a larger, more complex application is pending.

The initial experiments also revealed three important lessons: First, the component specification must be detailed enough to provide a relatively deterministic path to implementation, yet it should avoid encoding details that are technically important but marginal from the perspective of the business logic. Recurrent platform-specific details, such as how handles (e.g., to an event bus) or retained references are represented, are better covered by an opinionated skill than explicitly stated in individual component specifications. Second, it is difficult to constrain the design-review agent so that it could reliably conclude that a design is already acceptable. Without additional guardrails and explicit completion criteria, the review phase tends to continue identifying minor technical issues that do not materially affect architectural quality. Third, the relevant class of tasks cannot be delimited precisely enough to treat the skill as fixed. The platform-dependent skill must continue to evolve as new components and recurring platform patterns

are encountered. This is consistent with traditional software engineering practice, where the development process itself evolves across projects. We therefore expect frequent skill updates at the current stage, followed by a gradual slowdown as knowledge accumulates across a broader range of components.

Future work should specialize the COMPAID method to several additional domains and platforms. On that basis, a meta-model of the common platform-independent concepts should be derived, and the process by which those concepts are specialized for a concrete platform should be made more systematic.

References

1. —, Companion package, https://github.com/smartarch/COMPAID-companion-package.
2. Alphabet Investor Relations, 2024 Q3 Earnings Call, (2024). https://abc.xyz/2024-q3-earnings-call/ (visited on 03/22/2026)
3. Alphabet Investor Relations, 2025 Q1 Earnings Call, (2025). https://abc.xyz/2025-q1-earnings-call (visited on 03/22/2026)
4. Anthropic, Claude Code, (2026). https://claude.com/product/claude-code (visited on 03/27/2026)
5. Espressif Systems, ESP-IDF Programming Guide, (2026). https://docs.espressif.com/projects/esp-idf/en/stable/esp32/ (visited on 03/22/2026)
6. Huang, D., Zhang, J.M., Luck, M., *et al.*: AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation, (2024). arXiv: 2312.13010.
7. Islam, M.A., Ali, M.E., Parvez, M.R.: MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. In: Proceedings of ACL 2024 (Vol. 1). ACL (2024). https://doi.org/10.18653/v1/2024.acl-long.269
8. Jimenez, C.E., Yang, J., Wettig, A., *et al.*: SWE-bench: Can Language Models Resolve Real-world GitHub Issues? In: The Twelfth International Conference on Learning Representations (2024). https://openreview.net/forum?id=VTF8yNQM66
9. Maes, S.: Ensuring the Maintainability and Supportability of "Vibe-Coded" Software Systems: A Framework for Bridging Intuition and Engineering Rigor. (2025). https://doi.org/10.5281/ZENODO.15354102
10. OpenAI, Codex, (2026). https://openai.com/codex/ (visited on 03/27/2026)
11. OpenAI Help Center, Skills in ChatGPT, (2026). https://help.openai.com/en/articles/20001066-skills-in-chatgpt (visited on 03/27/2026)
12. Roman, A., Roman, J.: Orchestral AI: A Framework for Agent Orchestration, (2026). arXiv: 2601.02577.
13. Sapkota, R., Roumeliotis, K.I., Karkee, M.: Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI, (2025). arXiv: 2505.19443.
14. Steghöfer, J.-P., Borg, M.: An Abstraction Is Worth a Thousand Vibes. IEEE Software **43**(1) (2026). https://doi.org/10.1109/MS.2025.3621786
15. Zeff, M.: Microsoft CEO says up to 30% of the company's code was written by AI, TechCrunch. (2025). https://techcrunch.com/2025/04/29/microsoft-ceo-says-up-to-30-of-the-companys-code-was-written-by-ai/ (visited on 03/22/2026)