OPEN ACCESS

ARTICLE

INNOVATION PRESS

Journal homepage: www.the-innovation.org/ai-plus

AI Plus

AI Plus 1 (2026):100007

Contents lists available at INNOVATION PRESS

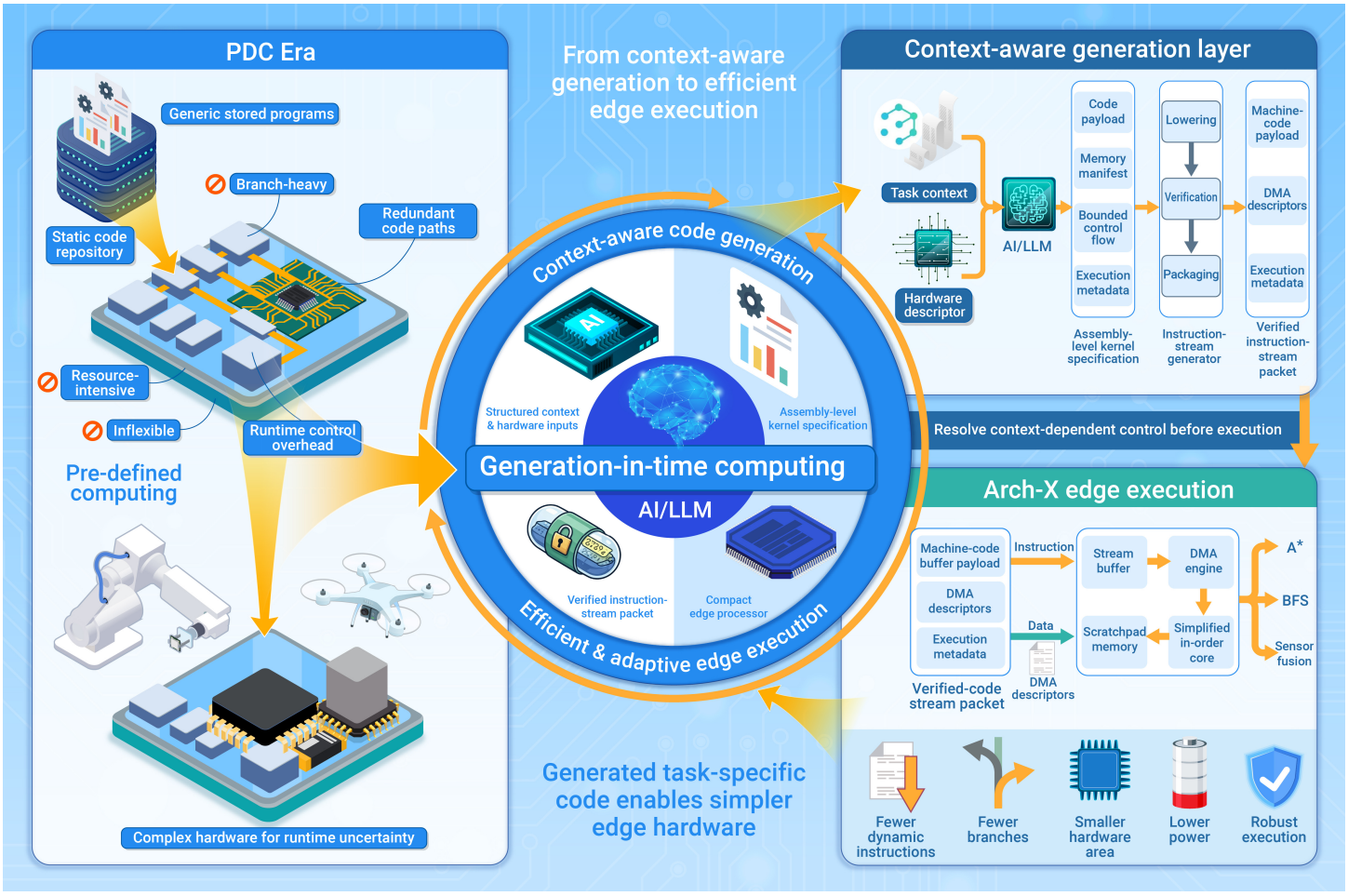
GTC: architecture exploration for generation-in-time computing paradigm

Pengju Ren,[1,2,*] Gelin Fu,[1,2] Shenghuan Liu,[1,2] Yadong Zhang,[1,2] Qiwei Dang,[1,2] Tian Xia,[1,2] Wenzhe Zhao,[1,2] and Nanning Zheng[1,2]

*Correspondence: *Correspondence: pengjuren@xjtu.edu.cn

Received: May 11, 2026; Accepted: August 15, 2026; Published Online: August 18, 2026; https://doi.org/10.59717/ipj.aiplus.2026.100007

GRAPHICAL ABSTRACT



PUBLIC SUMMARY

- Devices often store many predefined programs even when only one task is active at a time.
- Generation-in-Time Computing (GTC) builds context-specific machine-code streams before edge execution.
- GTC cuts dynamic instructions by 68.8% and FPGA logic power by 52.7% in a path-planning case study.
- GTC is especially promising for robots, sensors, and IoT systems that must adapt quickly to changing contexts.

ARTICLE

OPEN ACCESS

INNOVATION PRESS

Journal homepage: www.the-innovation.org/ai-plus

AI Plus

AI Plus 1 (2026):100007

Contents lists available at INNOVATION PRESS

GTC: architecture exploration for generation-in-time computing paradigm

Pengju Ren,[1,2,*] Gelin Fu,[1,2] Shenghuan Liu,[1,2] Yadong Zhang,[1,2] Qiwei Dang,[1,2] Tian Xia,[1,2] Wenzhe Zhao,[1,2] and Nanning Zheng[1,2]

[1]National Key Laboratory of Human-Machine Hybrid Augmented Intelligence, Xi'an Jiaotong University, Xi'an 710049, China

[2]Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Xi'an 710049, China

*Correspondence: *Correspondence: pengjuren@xjtu.edu.cn

Received: May 11, 2026; Accepted: August 15, 2026; Published Online: August 18, 2026; https://doi.org/10.59717/ipj.aiplus.2026.100007

Citation: Ren P., Fu G., Liu S., et al. (2026). GTC: architecture exploration for generation-in-time computing paradigm. *AI Plus* **1**:100007.

We propose Generation-in-Time Computing (GTC), a software-hardware co-design paradigm in which a centralized, large language model-based generation system translates real-time physical context and target hardware constraints into task-specific executable instruction streams. To realize this paradigm, we present Arch-X, a computation-centric edge architecture that exploits these specialized streams by eliminating or simplifying complex microarchitectural structures commonly used in high-performance processors. We evaluate GTC on three workloads: LLM-guided A* path planning, breadth-first search (BFS), and TinyEKF sensor fusion. For LLM-guided A*, GTC reduces the number of dynamic instructions by 68.8% and branch instructions by 81.9% relative to a predefined-computing baseline. Across all three workloads, GTC improves performance on both in-order and out-of-order processors. An FPGA implementation further shows that Arch-X reduces lookup-table (LUT) and block-RAM (BRAM) usage by 71.1% and 86.2%, respectively, relative to a dual-issue out-of-order reference core, while retaining the arithmetic datapath required by the evaluated workloads. Together, these results provide preliminary evidence that resolving context-dependent control through centralized generation can reduce software overhead and enable simpler edge hardware for efficient, adaptive computation.

INTRODUCTION

Conventional computing systems encode a wide range of possible behaviors before execution, even when the current context determines which behavior is actually needed. This model is effective for general-purpose computing but is poorly suited to systems that operate under changing tasks, data, and hardware conditions. At any given moment, such a system may need to perform only one specific task, while its stored program must support many possible scenarios. Consequently, both the software stack and the hardware may spend substantial resources selecting among alternatives that the available context has already ruled out.

We refer to this conventional model as Predefined Computing (PDC). A PDC system stores alternative code paths and the logic needed to select among them. During execution, the system identifies the path that matches the current context. Modern processors and software stacks are highly optimized for this process through branch prediction, instruction and data caching, dynamic scheduling, profile-guided specialization, and multiple layers of dispatch.[1–6] These mechanisms provide flexibility and high general-purpose performance. However, they also consume instruction bandwidth, memory capacity, energy, and hardware resources to manage runtime uncertainty rather than directly performing the active computation.

Generation-in-Time Computing (GTC) takes a different approach. When relevant context is available before a kernel begins, a generation system can use the current task and target-hardware constraints to produce a specialized executable stream. The resulting computation is therefore tailored to the current conditions rather than selected from a general program that represents many possible conditions.

In this paper, we formulate GTC as a paradigm for context-aware execution. A GTC generator receives a task-context description and a hardware descriptor. It then produces a compact executable stream, checks the stream against safety and resource constraints, and delivers it to the target hardware. The execution platform may be a conventional CPU or a domain-specific processor designed for generated kernels. GTC is not intended to generate all computation on demand. Instead, it targets workloads in which only a small part of a large task space is active, and the available context can resolve important control decisions before execution.

The key idea is to move some context-dependent control decisions from runtime execution to a centralized generator. A stream specialized to the current context can contain fewer branches, fewer inactive code paths, and more explicit data movement. This specialization creates an opportunity to simplify the execution hardware and devote a larger share of its resources to arithmetic and task-specific operations. Before execution, each generated stream is checked against architectural, memory, and timing constraints. GTC, therefore, executes bounded, verifiable kernels rather than unrestricted generated code.

Figure 1(a) illustrates the difference between PDC and GTC using a smart-home IoT gateway. In the PDC system, the gateway locally stores drivers and compiled routines for multiple devices and operating conditions. At runtime, it identifies the active device, selects the appropriate code path, and executes the corresponding routine. Examples include controlling an air conditioner based on indoor temperature or adjusting lighting based on ambient illumination. In the GTC system, a central generator receives the current state of the household devices and produces only the code required for the active scenario. For example, when the indoor temperature is high and an air conditioner is available, the generator produces a cooling-control routine and delivers it to the edge device as a compact instruction sequence. When the system supports many possible tasks but only a few are active at a time, this approach can reduce local code-storage requirements and the resources used for instruction retrieval and control.

Figure 1(b) generalizes this example into two coupled components: a central software stack that resolves task and hardware constraints, and a local execution engine that runs the resulting stream. The central LLM-based system translates high-level requirements, physical context, and target hardware constraints into short, executable kernels. A single generator can support heterogeneous devices with different instruction sets and architectural configurations, thereby moving part of the software and control complexity away from the edge. The local architecture, in turn, focuses on executing the generated stream. Because the stream is compact and task-specific, the processor can simplify structures used primarily for speculative control and general-purpose execution, including branch predictors, reorder buffers, instruction caches, and prefetching logic.[2,3,7–9] Software-managed local memory can also provide more predictable data access.

To investigate this paradigm, we present Arch-X, a configurable framework for exploring GTC-based edge architectures. Arch-X supports configurable datapath widths, functional-unit configurations, local-memory organizations, and instruction-delivery mechanisms. We use it to test whether resolving context-dependent control before execution can simplify both the

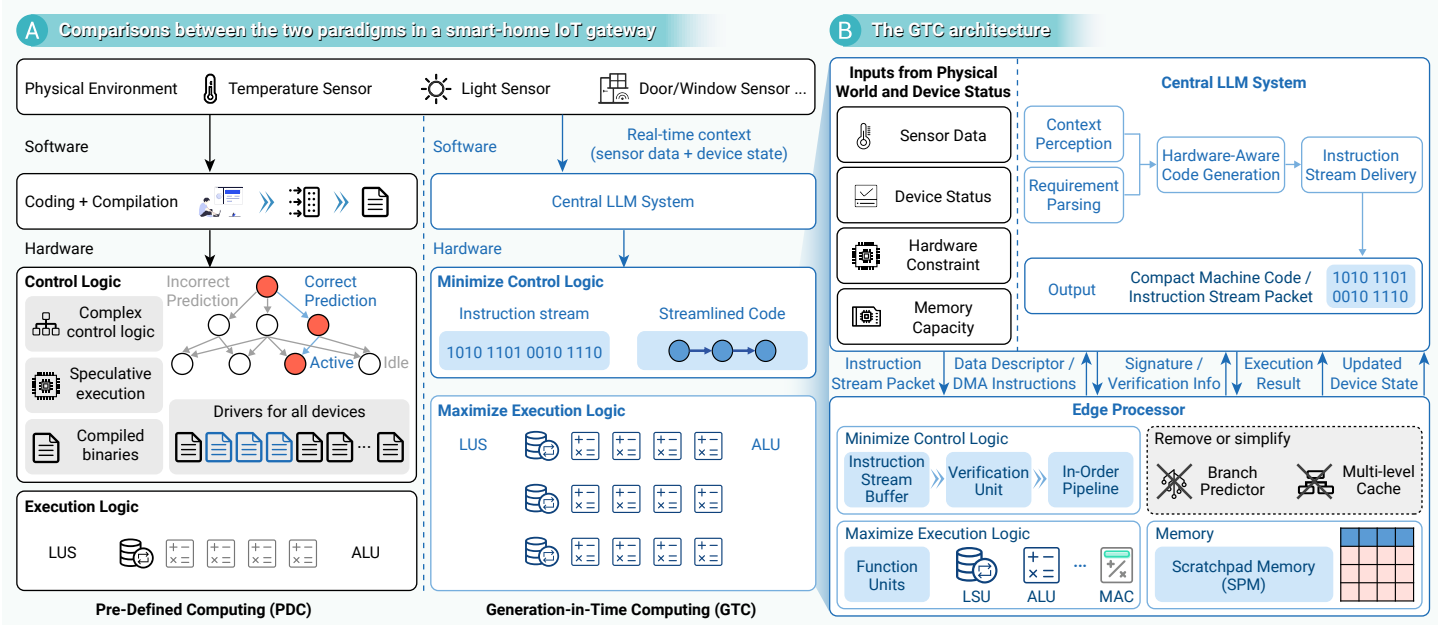


Figure 1. From Predefined Computing to Generation-in-Time Computing (a) Comparison of PDC and GTC in a smart-home IoT gateway. PDC stores compiled binaries and device routines locally, whereas GTC uses the current context to generate a compact, task-specific instruction stream. (b) Two-layer GTC architecture consisting of a central LLM-based software stack and a simplified processor connected through instruction-stream delivery.

instruction stream and the underlying microarchitecture. Our evaluation covers three workloads: LLM-guided A* path planning,[10,11] breadth-first search (BFS), and TinyEKF sensor fusion. For A* path planning, GTC reduces the number of dynamically fetched instructions by 68.8%, branch instructions by 81.9%, branch mispredictions by 97.6%, and the static code footprint by 97.7% relative to a precompiled baseline. GTC also improves the performance of BFS and TinyEKF on both in-order and out-of-order processors. In addition, an FPGA implementation of a simplified Arch-X core reduces LUT and BRAM usage by 71.1% and 86.2%, respectively, relative to a dual-issue out-of-order reference core, while retaining the arithmetic datapath required by the evaluated workloads. These results provide preliminary evidence that GTC can reduce software-control overhead and enable simpler hardware for context-specialized edge computing.

MATERIALS AND METHODS

LLM-BASED CODE GENERATION

Recent advances in LLMs provide the technical foundation for GTC. Modern models can interpret natural-language and structured requirements, synthesize code, reason about task constraints, and adapt programs to different software interfaces and instruction sets.[12–18] However, most LLM-based development tools operate above the hardware boundary. They produce software artifacts that are compiled and executed on conventional processors, thereby changing the software rather than the organization of the hardware that executes it.

In GTC, the LLM maps high-level intent and partially structured physical context to a hardware-aware kernel specification. This semantic mapping is difficult to capture with fixed templates when tasks and environments vary substantially. Deterministic tools then handle assembly and constraint enforcement, allowing the LLM to focus on semantic interpretation and code generation.

RUNTIME SPECIALIZATION AND PROGRAM GENERATION

We use Predefined Computing (PDC) as an analytical abstraction for systems in which the executable structure is largely determined before the current context is known. Conventional systems can still adapt through just-in-time compilation, runtime specialization, partial evaluation, and program synthesis.[19–23] These methods generally begin with a program structure defined by a developer or an earlier compiler pass.

GTC instead treats semantic context and hardware constraints as first-class inputs to executable-stream generation. The resulting object is a bounded, context-specialized executable rather than another version of a general-purpose application. This distinction does not depend solely on the use of an LLM; templates, compilers, or other generators may also participate when the context is sufficiently structured. The broader objective is to determine whether context-aware generation can redistribute work across the software–hardware boundary.

ARCHITECTURAL MOTIVATION

General-purpose processors use speculation, dynamic scheduling, and hardware-managed memory hierarchies to maintain performance when control flow and data access are not known in advance.[24,25] Short, bounded kernels with few inactive alternatives may reduce the demand for speculative control, dynamic scheduling, and hardware-managed data movement. These properties do not make complex microarchitectural structures unnecessary in general, but they may allow selected structures to be simplified for context-specialized workloads. GTC studies whether this regularity enables a different balance of hardware resources.

This opportunity is particularly relevant to physical AI and low-power edge systems, where each immediate action often activates only a small portion of a much larger task space.[26–28] GTC complements rather than replaces conventional execution and is most suitable when context can specialize a bounded computation before execution. Existing work has largely used generation to improve software running on conventional hardware. GTC extends this direction by asking whether context-aware executable generation and execution hardware can be co-designed for efficient, adaptive computation.

GTC PARADIGM

We define Generation-in-Time Computing (GTC) as an execution paradigm that uses context available before execution to generate a bounded executable for the active task.[29,30] The executable is generated, checked, delivered, and executed as part of a coordinated workflow. GTC complements predefined computing rather than replacing it. It targets cases in which the task space is large, only a small portion is currently active, and the available context can specialize the computation before execution.

A GTC system has two interacting layers. The generation layer maps semantic requirements and architectural constraints to an executable stream.[31,32] Semantic requirements specify the task, relevant input state, safety and timing constraints, and expected output. Architectural constraints describe the target instruction set, compute units, memory resources, interfaces, and resource limits. The same task may therefore produce different kernels for different devices, while the same device may receive different

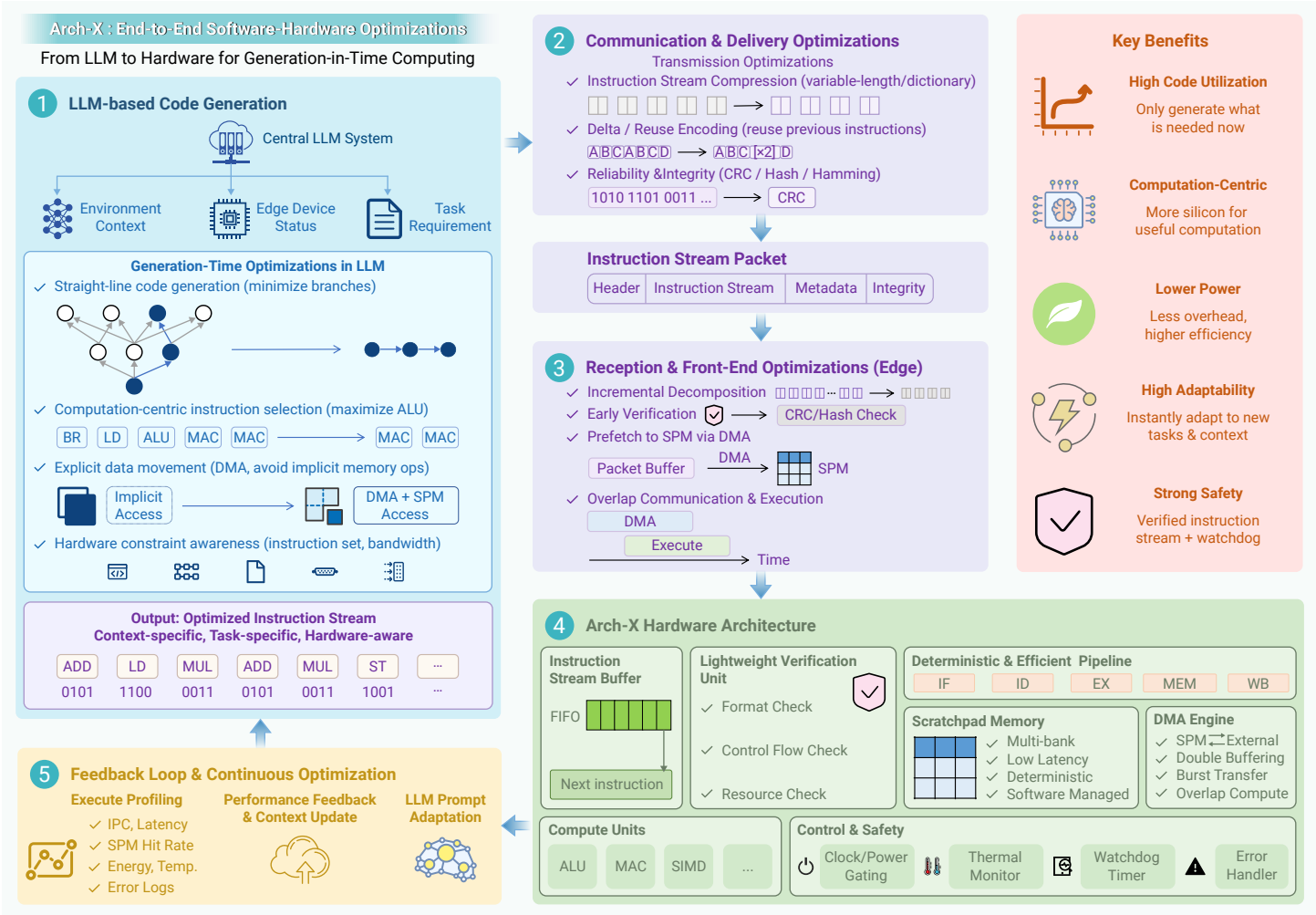


Figure 2. Cross-stack workflow of GTC The central system combines task context with hardware constraints and emits an instruction-stream packet containing code, data descriptors, and execution metadata. The execution engine stages data in scratchpad memory, runs the bounded kernel, and returns execution feedback.

kernels as its context changes.

The execution layer checks the generated kernel, stages its data, executes the instruction stream, and returns the result or a status record. This layer may be a conventional processor or a specialized execution engine. In either case, part of the task selection and specialization has already occurred in the generation layer.[33] When this process produces kernels with small instruction footprints, low branch density, explicit data requirements, and predictable control flow, the execution hardware may be simplified accordingly.

DESIGN GOAL

GTC has two related design goals. At the software level, it aims to reduce instruction overhead that does not contribute directly to the active task. At the hardware level, it aims to reduce the support structures required for general-purpose execution when bounded, specialized kernels do not benefit substantially from them.

The software goal is to reduce generic dispatch, inactive code paths, and runtime parameter handling while preserving the operations required for correctness and safety. The generator uses the current context and hardware description to resolve selected control-flow and data-placement decisions before execution. The resulting kernel can be smaller and execute fewer instructions than a general implementation. We evaluate this goal using metrics such as dynamic instruction count, branch count, and static code footprint rather than assigning individual instructions to subjective "useful" and "non-useful" categories.

The hardware goal is to reduce the cost of structures that provide general-purpose performance but offer limited benefits for the generated kernels. Branch predictors, dynamic schedulers, caches, and related structures remain valuable for workloads with unpredictable control flow and data access. However, selected structures may be reduced or omitted when kernels are bounded and their data movement is explicit. We evaluate this goal using FPGA resource and power measurements, including LUT, register, BRAM, logic power, and signal power results.

The two goals are coupled. Lower branch density may reduce the benefit of aggressive branch prediction, while an explicitly declared working set may reduce reliance on cache discovery and prefetching.[34,35] Scheduling for a known datapath may likewise reduce the benefit of dynamic instruction reordering. These effects depend on the workload and generated stream; they are not universal properties of GTC. The central hypothesis is that, for suitable bounded kernels, generation can reduce both dynamic instruction overhead and the hardware support needed to execute the remaining computation.

GTC WORKFLOW

Figure 2 shows the GTC workflow from context acquisition to execution feedback. First, sensor summaries, device telemetry, and application requests are combined into a structured context record. This record identifies the active task and the constraints relevant to generation.[36] It contains task-level information rather than all raw input data; bulk data remain in memory buffers, sensors, or external storage.

Second, the context record is paired with a hardware descriptor that specifies the target ISA, available compute units, memory resources, interfaces, and execution constraints.[37,38] Together, these inputs form a generation request that states what must be computed, which resources are available, and which safety and timing constraints apply. This explicit request constrains generation rather than treating it as an open-ended natural-language task.[39,40]

Third, the central system generates a task-specific kernel for the request.[32,41] In our prototype, the LLM emits a structured, hardware-aware specification whose code payload is expressed in target-ISA assembly. A deterministic assembler converts this payload into machine-code bytes. Values such as local map dimensions, waypoint coordinates, buffer offsets, and loop bounds can be fixed during generation, thereby eliminating the need for generic parameter handling and inactive control paths. The result is a compact executable for the active task rather than a reusable general-purpose application.

Fourth, the machine-code payload is combined with DMA descriptors and execution metadata to form an instruction-stream packet. The packet includes a task identifier, sequence number, execution bounds, and integrity metadata. Before delivery, deterministic checks enforce ISA legality, declared branch targets, memory ranges, and control-flow bounds.[30,42] These checks do not prove functional correctness or guarantee safety. They restrict the stream to a policy-constrained executable envelope and prevent the execution layer from treating LLM-generated declarations as trusted proof.

Fifth, the edge device executes the packet as a streamed kernel. In the evaluated Arch-X prototype, the instruction-stream interface buffers the code, DMA stages the working set in scratchpad memory, and an in-order pipeline fetches instructions from the stream buffer. Arch-X omits or simplifies aggressive instruction caching, dynamic branch prediction, and out-of-order scheduling because the evaluated kernels have bounded control flow and explicit data placement. A watchdog enforces the declared execution bound and triggers a fallback when the packet exceeds its permitted execution window.

After execution, the device returns the result together with status and deviation information. Repeated contexts can reuse previously checked templates, while new or changed contexts can trigger partial or full regeneration.[43] Reuse places GTC on a continuum between fully predefined and fully generated execution and can reduce repeated generation overhead.

APPLICABILITY AND LIMITATIONS

GTC targets sparse activation within a large space of possible computations. This regime has three properties: many routines are possible, only a small subset is active in the current context, and the context contains enough information to specialize that subset before execution.[44] GTC converts this contextual information into an executable specialized for the active case.

This pattern may arise in edge intelligence, where the physical state determines the active sensing, planning, or control kernel.[36,45] It may also occur in data analytics, scientific computing, and heterogeneous acceleration, where data properties and hardware capabilities determine the appropriate operator or kernel implementation.[41,46] These domains differ, but each may expose a small active computation within a larger set of possible behaviors.

GTC is most suitable when the active computation can be expressed as a bounded kernel, the context is sufficient for specialization, and generation latency can be hidden or amortized through reuse. The generated stream must also be checkable against memory, control-flow, timing, and output constraints. GTC is less suitable for stable workloads already served efficiently by predefined binaries, for tasks that cannot be constrained through a bounded interface, or when context changes faster than code can be generated and delivered. These conditions define GTC as a complementary execution mode rather than a universal replacement for predefined computing.

RESULTS

We evaluate GTC using Arch-X, a software–hardware system in which context-aware generation produces task-specific instruction streams for a specialized execution engine. LLM-guided A* path planning is our primary workload because it illustrates both layers of GTC: high-level reasoning selects the search region, and low-level generation produces a bounded local-search kernel. We also evaluate breadth-first search (BFS) and TinyEKF sensor fusion to determine whether the benefits extend beyond path planning. For each workload, the PDC baseline uses a generic implementation that supports multiple task instances, whereas GTC specializes the instruction stream for the current context and target hardware. Figure 3 summarizes the mapping of LLM-A* to the Arch-X software and hardware stack.

WORKLOAD: LLM-GUIDED A* PATH PLANNING

A* is an informed-search algorithm for finding the lowest-cost path in a weighted graph.[11] At each iteration, it expands the node n with the smallest evaluation value, $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the start node and $h(n)$ estimates the remaining cost to the goal. A* is a useful test case for GTC because its mathematical core is stable, whereas practical implementations vary with map size, obstacle placement, target position, and data structure organization.

In LLM-A*,[10] a first LLM-assisted stage derives high-level waypoints $T = \{t_1, t_2, \ldots\}$ from the environment. A second generation stage uses the current waypoint, local obstacle state, and hardware descriptor to produce a local-search kernel. The execution layer then runs A* between successive waypoints using staged local data. This division separates contextual planning from the bounded search performed by the edge hardware.

ARCH-X SOFTWARE

Arch-X implements the two LLM-assisted stages described above. The first produces high-level waypoints, and the second generates a bounded assembly payload for the current local-search instance. A deterministic assembler converts the payload into machine-code bytes, after which a stream-delivery module attaches DMA descriptors and execution metadata.

The software stack contains an environment manager, an LLM inference engine, an instruction-stream generator, and the stream-delivery module. The environment manager tracks the map, obstacle state, and robot pose. The inference engine produces intermediate waypoints, while the instruction-stream generator combines the current waypoint, local obstacle window, and Arch-X hardware descriptor to generate the kernel. Known grid dimensions, waypoint coordinates, and loop bounds are encoded as immediate values or fixed scratchpad offsets. The grid, open set, closed set, and result buffer use a static memory layout, and DMA descriptors specify data movement.

The LLM produces a structured kernel specification containing target-ISA assembly, memory and control-flow descriptors, resource declarations, and verification metadata. A deterministic packager constructs the packet header, encodes the assembly payload, and attaches the data descriptors. The resulting packet is specialized for one local planning instance and omits much of the function-call, parameter-passing, and inactive-control-flow overhead of the generic implementation.

Table 1 shows that GTC reduces the number of dynamically fetched instructions by 68.8% and branch instructions by 81.9%. These reductions are consistent with the removal of generic control and data-structure operations. Branch mispredictions decrease from 41 to 1, and the evaluated code footprint decreases from 8,440 to 196 bytes. GTC incurs no data-cache misses because DMA bypasses the cache and explicitly stages the working set; this result does not include the cost of DMA transfers.

BFS AND SENSOR-FUSION WORKLOADS

We use BFS and TinyEKF to evaluate whether GTC's performance benefits extend to graph processing and sensor fusion. Table 2 compares PDC and GTC on both in-order and out-of-order processors. Each speedup is computed relative to PDC on the same processor type.

For BFS, the PDC implementation retains generic graph construction, queue management, and runtime control paths, whereas GTC specializes the instruction stream to the current graph and hardware constraints. GTC achieves a 3.2× speedup on the in-order processor and a 2.6× speedup on the out-of-order processor.

We also evaluate a sensor-fusion kernel based on TinyEKF.[59] The PDC implementation retains generic update logic for multiple sensor configurations and runtime conditions. In GTC, the task context fixes the active sensor combination, state dimension, measurement layout, and update path before execution. GTC achieves a 3.3× speedup on the in-order processor and a 2.7× speedup on the out-of-order processor.

CORRECTNESS AND ROBUSTNESS

In GTC, the generation request contains the task context, hardware descriptor, memory layout, ISA constraints, safety requirements, and required output schema. We use Cursor as the LLM-assisted generation environment. The evaluated LLMs include GPT-5.5, Fable 5, Gemini 3.1 Pro, and Kimi K2.5.

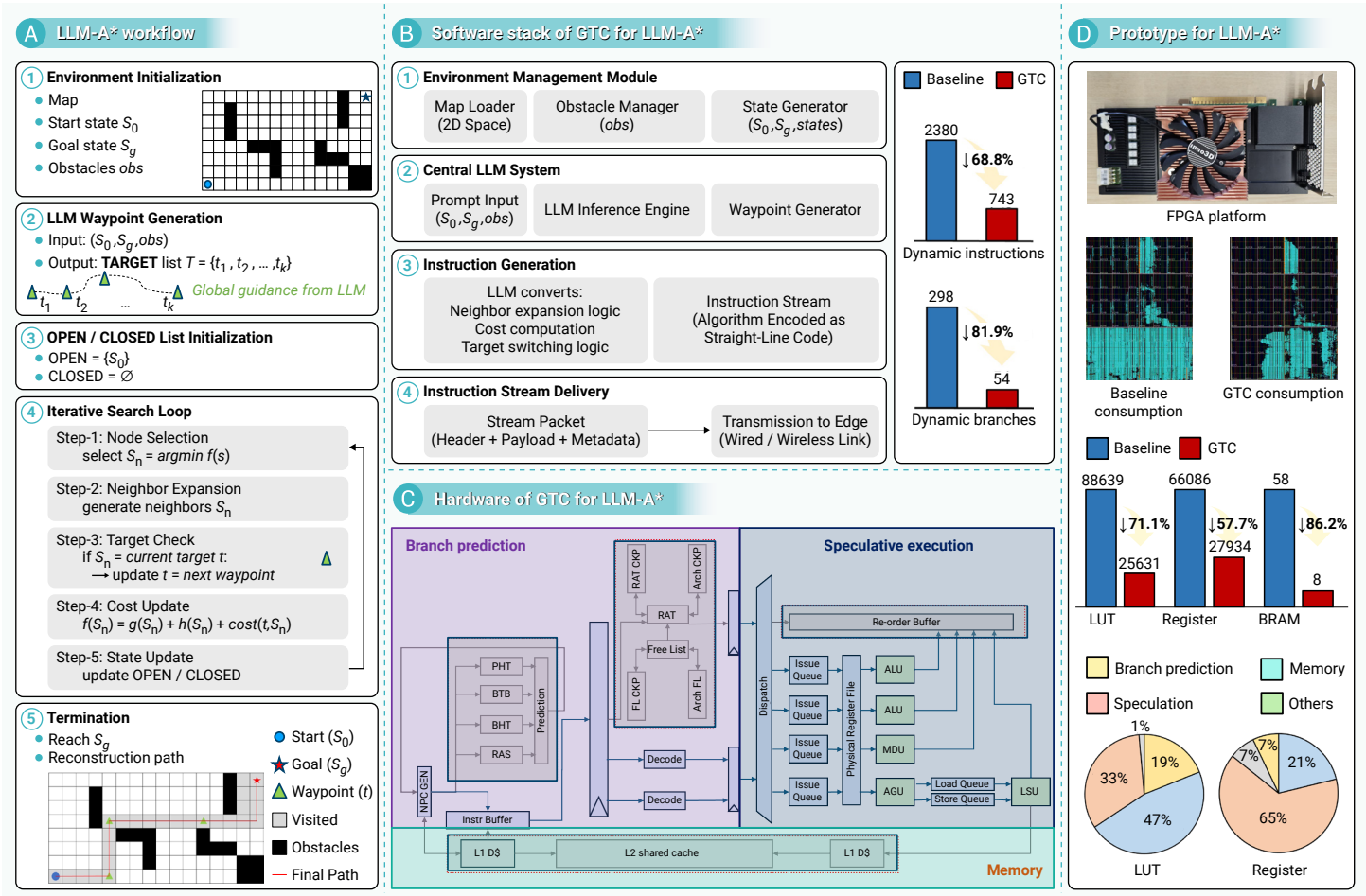


Figure 3. Mapping LLM-A* onto Arch-X (a) LLM-A* decomposes path planning into waypoint generation and local A* search. (b) The Arch-X software stack converts the planning context and hardware descriptor into a compact instruction-stream packet. (c) The Arch-X hardware executes the stream on a simplified microarchitecture. (d) Software measurements quantify instruction-level effects, while FPGA implementations measure hardware resource reductions.

Table 1. Instruction-level comparison between the PDC baseline and GTC.

Metric	PDC Baseline	GTC	Reduction
Dynamically fetched instructions	2,380	743	68.8%
Dynamic branch instructions	298 (12.5%)	54 (7.3%)	81.9%
Branch mispredictions	41	1	97.6%
Data-cache misses	35	0 (DMA bypass)	–
Static code footprint (bytes)	8,440	196	97.7%

Table 2. Cross-workload speedup of GTC relative to PDC on the same processor type.

Workload	In-order speedup	Out-of-order speedup
LLM-A*	3.5×	2.8×
BFS	3.2×	2.6×
TinyEKF	3.3×	2.7×

Table 3. Cross-model robustness on the BFS workload. Performance is normalized to GPT-5.5.

Model	GPT-5.5	Fable 5	Gemini 3.1 Pro	Kimi K2.5
Normalized performance	1.00×	1.09×	1.03×	0.85×

For correctness evaluation, we run the complete GTC workflow across 100 independent LLM-A* scenarios with varying task contexts, including campus delivery, data center maintenance, hospital evacuation, and museum security. A run is counted as correct only when the final execution result satisfies the task requirement and produces a valid collision-free path to the target. The results show that the success rate is 89%.

To assess model robustness, we evaluate the BFS workload using the same GTC generation request across multiple LLMs. Performance is normalized to GPT-5.5 (1.0). As shown in Table 3, the generated kernels remain within a relatively narrow performance range across models, suggesting that

Table 4. Ablation for LLM generation, memory organization, and branch-control hardware.

Variant	Change from GTC reference	Speedups
GTC	LLM generation with DMA/scratchpad and in-order core	1.00×
GTC w/o LLM generation	Replace LLM with conventional PDC	0.30×
GTC w/o DMA/scratchpad	Replace DMA/scratchpad with cache-based memory organization	0.60×
GTC w/ dynamic branch predictor	Add dynamic branch prediction	1.06×

Table 5. Controlled comparison of PDC and GTC on in-order and out-of-order processors. Performance is normalized to PDC on the in-order processor for each workload.

Workload	PDC+in-order	GTC+in-order	PDC+out-of-order	GTC+out-of-order
A*	1.0×	3.5×	1.5×	4.3×
BFS	1.0×	3.2×	1.4×	3.9×
TinyEKF	1.0×	3.3×	1.5×	4.2×

Table 6. Hardware simplifications in Arch-X.

Hardware component	Representative structures	Arch-X optimization
Branch prediction	PHT, BTB, BHT, RAS	Simplified by static prediction
Register renaming and scheduling	RAT, free list, IQ, ROB	Removed or replaced by architectural register file
Cache and load/store system	L1 cache, L2 cache, LSQ	Simplified using stream buffer, scratchpad, and DMA
Execution datapath	Decode, ALU, MDU, LSU	Retained

Table 7. FPGA resource reduction compared to the baseline.

Resource	Baseline	Arch-X	Reduction: Amount / Percentage				
			Total	Branch prediction	Speculation	Memory	Others
LUT	88639	25631	63008/71.1%	4493/7.1%	13342/21.2%	40677/64.6%	4496/7.1%
Register	66086	27934	38152/57.7%	7220/18.9%	17846/46.8%	12545/32.9%	541/1.4%
BRAM	58	8	50/86.2%	0/0%	0/0%	50/100%	0/0%

the proposed GTC interface is not tied to a single LLM.

4.5 ABLATION AND CONTROLLED HARDWARE COMPARISON

We further analyze the effects of LLM generation, memory organization, and hardware simplification using the BFS workload. As shown in Table 4, to isolate the contribution of LLM-based generation and code specialization, we replace the GTC-generated specialized stream with the conventional PDC implementation while keeping the same in-order hardware, DMA, and scratchpad memory. This results in a 70% performance slowdown compared with the GTC reference, indicating that context-specialized code generation is a major contributor to performance gains. To isolate DMA staging and scratchpad memory, we replace the DMA/scratchpad organization with a cache-based memory structure while keeping the GTC-generated code and the same in-order execution core. This variant achieves only 60% of the reference performance, showing that explicit data staging and software-managed local memory are important for GTC execution. Finally, adding a dynamic branch predictor to the reference configuration improves performance by only 1.06×, suggesting that additional control hardware provides limited benefit for the compact generated streams.

For a fair hardware comparison, we evaluate both PDC and GTC on in-order and out-of-order processor configurations. Table 5 reports performance normalized to the PDC implementation on the in-order processor. GTC improves performance on both processors. Across the three evaluated workloads, GTC achieves an average speedup of 3.3× on the in-order processor compared with 2.8× on the out-of-order processor, indicating that its performance benefit is more pronounced on the simpler in-order architecture because the generated instruction streams contain fewer instructions and fewer branch operations. This also indicates that the gains do not result from replacing an out-of-order core with a simpler in-order design. The area reduction primarily comes from the simplified in-order microarchitecture, while GTC enables this simpler architecture to retain strong task-level performance by reducing runtime control and instruction-processing overhead.

ARCH-X HARDWARE

Arch-X is designed for the regular instruction streams produced by the software stack. Its instruction-stream interface validates packet headers and places the code payload in a small stream buffer. Before execution, the DMA engine transfers the required data into scratchpad memory. A simplified in-order core then executes the kernel while retaining the arithmetic and load/store operations required by the evaluated workloads.

The evaluated streams have low branch density, fixed instruction ordering, and explicit data placement. Arch-X therefore uses static branch prediction, omits register renaming and dynamic scheduling, and replaces cache-based instruction delivery with a stream buffer. Scratchpad memory and DMA provide software-managed data movement. Table 6 summarizes these changes.

Arch-X retains the arithmetic functionality required by the evaluated kernels, but the removed structures may reduce performance on general-purpose workloads. Figure 3(c) illustrates the retained datapath and the structures simplified for streamed execution. This design trades some general-purpose flexibility for lower hardware overhead on the target workloads.

FPGA RESULTS

We use the Xilinx XCVU13P board as the FPGA prototype platform. We implement the simplified Arch-X hardware on an FPGA and compare it with an in-house RISC-V high-performance CPU baseline. The baseline includes conventional prediction, scheduling, and hierarchy structures, whereas Arch-X retains the decode, ALU, multiplier/divider, and load/store datapath while removing the structures, as shown in Table 7.

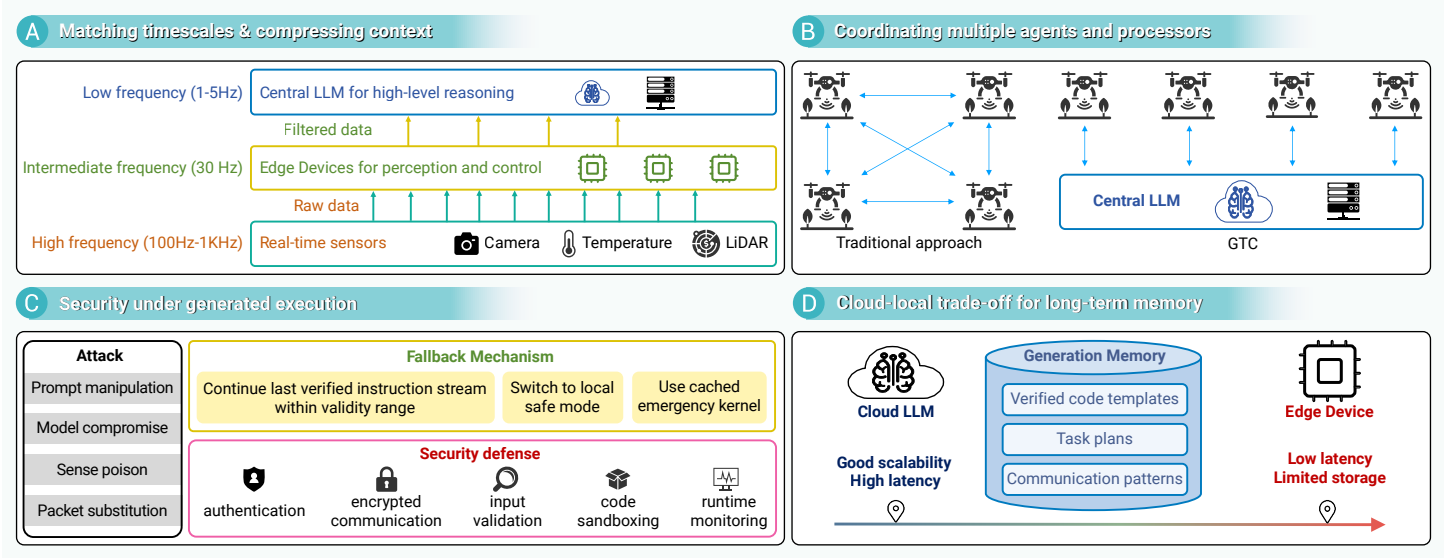


Figure 4. System-level design considerations for GTC (a) Generation responds to task-level context changes, while edge devices summarize raw sensor data. (b) A central generator may coordinate multiple execution units through a shared task state. (c) Generated execution introduces security concerns involving context, model behavior, and executable streams. (d) Long-term kernel storage ranges from centralized repositories to processor-local caches.

The FPGA results quantify the hardware consequences of generation-time specialization. LUT usage is reduced by 71.1% and register usage by 57.7%, reflecting the removal of prediction logic, speculative states, and dynamic scheduling structures. BRAM usage falls by 86.2%, indicating that hierarchy and buffering account for a substantial share of the baseline storage cost. The resulting processor allocates fewer resources to branch prediction, dynamic scheduling, and cache, with a large fraction of the remaining design dedicated to executing the generated kernel.

To complement the resource analysis, we also analyze the post-implementation Vivado power report, focusing on the Logic and Signal components, which capture the dynamic power associated with core logic activity and interconnect switching under the same FPGA and clock settings. The logic power is reduced from 0.74 W to 0.35 W, and the signal power is reduced from 0.82 W to 0.51 W, indicating that Arch-X not only reduces resource usage but also lowers the dynamic power overhead of non-memory support logic and routing.

DISCUSSION

GTC introduces system-level challenges that are not captured by instruction counts or FPGA resource measurements. These challenges include generation latency, context construction, multi-device coordination, security, and kernel reuse. Figure 4 summarizes these design considerations.

5.1 MATCHING GENERATION TO PHYSICAL TIMESCALES

A central timing question for GTC is how often the task-level context changes. Sensor sampling and low-level control may operate at hundreds of hertz or faster, whereas kernel generation is triggered only by changes in the task or environment. Therefore, fast stabilization and safety control remain local. For example, a robot can maintain motor control while the generator updates a waypoint after detecting a new obstacle, goal, or failure condition.[47] Figure 4(a) illustrates this separation among sensing, local execution, and generation.

Generation latency can be hidden only when the next kernel can be produced asynchronously and remains valid when execution begins. Kernel validity duration is therefore as important as model inference latency.[48,49] Each packet should carry a sequence number and validity interval so that the edge device can reject stale streams. If no valid stream is available, the device must use a previously checked kernel or a local fallback routine. Reusing cached kernels for recurring contexts can further reduce generation overhead.

RAW SENSOR DATA PROCESSING BEFORE GENERATION

Cameras, microphones, lidars, and other sensors can produce data faster than an edge device can transmit it to a remote generator. Therefore, GTC relies on local preprocessing and summarization.[50,51] The edge device sends task-level information, such as obstacle regions, rather than complete lidar frames. Bulk input data remain local and are accessed by the generated kernel when needed.

Local summarization can use filtering, window aggregation, feature extraction, anomaly detection, or confidence estimation. Because this process may discard information, the context record should include uncertainty or confidence when relevant. Routine states may use compact summaries, whereas uncertain or safety-critical states may require richer data or a conservative local routine.

COORDINATION OF MULTIPLE AGENTS AND PROCESSORS

A potential extension of GTC is the coordination of robot fleets, distributed sensors, or processors in a heterogeneous system.[52,53] A central generator could use a shared task state to produce local kernels that encode action order, sensing assignments, data ownership, or synchronization points. This approach treats the generator as a scheduler of bounded tasks rather than as a controller of every low-level action.

Centralized coordination could reduce some peer-to-peer communication, but it may also introduce latency, a bottleneck, and a single point of failure. Stale or incomplete global state can produce conflicting local actions. Sequence numbers, validity intervals, synchronization barriers, and deviation reports can help detect these conditions. During a network failure or timeout, each device should reject stale packets and fall back to a cached, previously checked kernel or a locally stored PDC routine. GTC execution can resume after a new stream passes integrity, memory-range, and control-flow checks.

SECURITY FOR GTC

Bounded kernels may reduce the attack surface when their memory access, control flow, and privileges are independently constrained.[54,55] This benefit is conditional: a short generated kernel can still contain unsafe memory accesses or incorrect control flow. Therefore, kernel size alone does not provide a security guarantee.

GTC also introduces risks related to the generator and the delivery path. Prompt injection, sensor poisoning, model compromise, packet substitution, and stale context can all affect the executable stream.[56] Therefore, a deployment requires packet authentication and integrity checks, freshness validation, memory isolation, control-flow constraints, least-privilege execution, and a local fail-safe path. These controls shift trust away from the LLM output and toward independently enforced execution policies.

CLOUD–LOCAL TRADE-OFF

Repeated use can produce a repository of previously checked kernels, data-layout templates, scheduling patterns, and context-to-code mappings.[43] A

centralized repository can share updates across devices, while a processor-local cache provides lower latency and remains available during disconnection.[57,58] Hybrid designs can retain frequently used or safety-critical kernels locally while sharing less common entries through the central generator.

Repository design must also address kernel versioning, hardware and firmware compatibility, revocation, stale context, and the privacy of stored task records. These requirements determine when a cached kernel remains valid and which device may reuse it. Figure 4(d) illustrates the continuum from centralized repositories to processor-local caches.

CONCLUSION

This paper introduces Generation-in-Time Computing (GTC), a software–hardware co-design paradigm that addresses the growing mismatch between static binary deployment and the sparse, context-dependent nature of modern edge workloads. Instead of storing pre-compiled code that must cover every possible scenario, GTC treats computation as an on-demand service in which a central generation system, built around a large language model, infers real-time physical context and hardware constraints, producing short, task-specific machine-code streams delivered to the edge. To execute such streams efficiently, we propose a configurable computation-centric processor that separates centralized control generation from local data execution and removes the speculative and hierarchical structures that conventional cores use to tolerate runtime uncertainty. GTC shows that when context-dependent control is resolved before execution, both the executed instruction stream and the supporting microarchitecture can be measurably simplified. Several directions remain open, including tighter integration between code generation and on-device verification, as well as end-to-end characterization under realistic communication conditions. We view GTC as a first step toward intelligent systems where centralized intelligence and computation-centric hardware jointly enable efficient, adaptive, and trustworthy execution.

DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work, the authors used Cursor in order to improve language and readability. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

REFERENCES

1. Smith J.E. (1998). A study of branch prediction strategies. *25 Years of the international symposia on Computer architecture (selected papers)* (*Association for Computing Machinery*), pp: 202–215. DOI: 10.1145/285930.285980
2. Tomasulo R.M.. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* **11**:25–33. DOI:10.1147/rd.111.0025
3. Smith J.E. and Pleszkun A.R.. (1988). Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* **37**:562–573. DOI:10.1109/12.4607
4. Pettis K. and Hansen R.C.. (1990). Profile guided code positioning. *SIGPLAN Not.* **25**:16–27. DOI:10.1145/93548.93550
5. Kessler R.E.. (1999). The Alpha 21264 microprocessor. *IEEE Micro* **19**:24–36. DOI:10.1109/40.755465
6. Yeager K.C.. (1996). The Mips R10000 superscalar microprocessor. *IEEE Micro* **16**:28–41. DOI:10.1109/40.491460
7. McFarling S. (1993). Combining branch predictors. (*Digital Western Research Laboratory*).
8. Smith A.J. and Goodman J.R.. (1985). Instruction cache replacement policies and organizations. *IEEE Trans. Comput.* **C–34**:234–241. DOI:10.1109/tc.1985.1676566
9. Smith A.J.. (1978). Sequential program prefetching in memory hierarchies. *Computer* **11**:7–21. DOI:10.1109/c-m.1978.218016
10. Meng S., Wang Y., Yang C.F. et al. (2024). LLM-A*: large language model enhanced incremental heuristic search on path planning. *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp: 1087–1102. DOI: 10.18653/v1/2024.findings-emnlp.60
11. Hart P.E., Nilsson N.J. and Raphael B.. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* **4**:100–107. DOI:10.1109/tssc.1968.300136
12. Jiang J., Wang F., Shen J. et al.. (2026). A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.* **35**:1–72. DOI:10.1145/3747588
13. Zhang S., Zhao J., Yu Q. et al.. (2026). The new compiler stack: A survey on the synergy of LLMs and compilers. *CCF Trans. High Perform. Comput.* **8**:148–179. DOI:10.1007/s42514-025-00270-x
14. Huynh N. and Lin B. (2025). Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint*. DOI: 10.48550/arXiv.2503.01245
15. Dakhel A.M., Majdinasab V., Nikanjam A. et al.. (2023). GitHub Copilot AI pair programmer: Asset or Liability?. *J. Syst. Softw.* **203**:111734. DOI:10.1016/j.jss.2023.111734
16. Yang J., Liu X., Lv W. et al. (2025). From code foundation models to agents and applications: A comprehensive survey and practical guide to code intelligence. *arXiv preprint*. DOI: 10.48550/arXiv.2511.18538
17. Gong J., Voskanyan V., Brookes P. et al. (2025). Language models for code optimization: Survey, challenges and future directions. *arXiv preprint*. DOI: 10.48550/arXiv.2501.01277
18. Romero Rosas M.R., Torres Sanchez M.A., and Eigenmann R. (2025). Should AI optimize your code? A comparative study of classical optimizing compilers versus current large language models. *Proceedings of the 2025 Supercomputing Asia Conference (ACM)*, pp: 22-29. DOI: 10.1145/3718350.3718357
19. Anand V., Garg D. and Kaufmann A. (2025). Iridescent: A framework enabling online system implementation specialization. *arXiv preprint*. DOI: 10.48550/arXiv.2508.16690
20. Jones N.D., Gomard C.K. and Sestoft P. (1993). Partial evaluation and automatic program generation. (*Peter Sestoft*).
21. de Assis Costa I.R., Santos H.N., Alves P.R. et al.. (2014). Just-in-time value specialization. *Comput. Lang. Syst. Struct.* **40**:37–52. DOI:10.1016/j.cl.2013.11.001
22. Aycock J.. (2003). A brief history of just-in-time. *ACM Comput. Surv.* **35**:97–113. DOI:10.1145/857076.857077
23. Kobaladze Z., Arnania A. and Sanikidze T. (2025). From provable correctness to probabilistic generation: A comparative review of program synthesis paradigms. *arXiv preprint*. DOI: 10.48550/arXiv.2508.00013
24. Mittal S.. (2018). A survey of techniques for dynamic branch prediction. *Concurr. Comput. Pract. Exp.* **31**:e4666. DOI:10.1002/cpe.4666
25. Banakar R., Steinke S., Lee B.S. et al. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proceedings of the tenth international symposium on Hardware/software codesign* (*Association for Computing Machinery*), pp: 73–78. DOI: 10.1145/774789.774805
26. Guan W., Hu Q., Li A. et al. (2025). Efficient vision-language-action models for embodied manipulation: A systematic survey. *arXiv preprint*. DOI: 10.48550/arXiv.2510.17111
27. Li X., He X., Zhang L. et al. (2025). A comprehensive survey on world models for embodied AI. *arXiv preprint*. DOI: 10.48550/arXiv.2510.16732
28. Liu F., Tang G., Li Y. et al.. (2019). A survey on edge computing systems and tools. *Proc. IEEE* **107**:1537–1562. DOI:10.1109/JPROC.2019.2920341
29. Wen X., Zhao Y., Xu X. et al. (2026). From craft to kernel: A governance-first execution architecture and semantic ISA for agentic computers. *arXiv preprint*. DOI: 10.48550/arXiv.2604.18652
30. McCann A.L. (2026). Governed metaprogramming for intelligent systems: Reclassifying eval as a governed effect. *arXiv preprint*. DOI: 10.48550/arXiv.2605.05248
31. Liu Z., Chen X., Wu H. et al.. (2025). Integrated sensing and edge AI: Realizing intelligent perception in 6G. *IEEE Commun. Surv. Tutor.* **28**:2725–2770. DOI:10.1109/COMST.2025.3592989
32. Nie J., Wu H., Lai Y. et al. (2026). KernelCraft: Benchmarking for agentic close-to-metal kernel generation on emerging hardware. *arXiv preprint*. DOI: 10.48550/arXiv.2603.08721
33. Dehury C.K., Kushwaha S.S., Zhang Q. et al. (2026). LLM-assisted Agentic Edge Intelligence Framework. *arXiv preprint*. DOI: 10.48550/arXiv.2604.09607
34. Wehmeyer L. and Marwedel P. (2005). Influence of memory hierarchies on predictability for time constrained embedded software. *Design, Automation and Test in Europe (IEEE)*, pp: 600–605. DOI: 10.1109/DATE.2005.183
35. Horro M., Rodríguez G., Touriño J. et al. (2018). Architectural exploration of heterogeneous memory systems. *arXiv preprint*. DOI: 10.48550/arXiv.1810.12573
36. Zhang Y., Malawade A.V., Zhang X. et al. (2023). CARMA: Context-aware runtime reconfiguration for energy-efficient sensor fusion. *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED) (IEEE)*, pp: 1-6. DOI: 10.1109/ISLPED58423.2023.10244517
37. Jain D., Pardeshi A., Frigo M. et al. (2025). ACT: Automatically generating compiler backends from tensor accelerator ISA descriptions. *arXiv preprint*. DOI: 10.48550/arXiv.2510.09932
38. Kan S. and Ertel S. (2026). Interaction tree semantics for RISC-V: Bridging compiler and hardware verification. *arXiv preprint*. DOI: 10.48550/arXiv.2605.04933
39. Li L., Rahili S. and Zhao Y. (2025). Correctness-guaranteed code generation via constrained decoding. *arXiv preprint*. DOI: 10.48550/arXiv.2508.15866
40. Mündler N., He J., Wang H. et al.. (2025). Type-constrained code generation with language models. *Proc. ACM Program. Lang.* **9**:601–626. DOI:10.1145/3729274
41. Liao G., Qin H., Wang Y. et al. (2025). KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at meta. *arXiv preprint*. DOI: 10.48550/arXiv.2512.23236
42. Mukherjee P., Lu M. and Delaware B. (2025). LLM-assisted synthesis of high-assurance c programs. *2025 40th IEEE/ACM International Conference on Automated*

Software Engineering (ASE), pp: 3108–3108. DOI: 10.1109/ase63991.2025.00255
43. Du W., Zhuo J., Dong Y. et al. (2026). AdaExplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation. *arXiv preprint*. DOI: 10.48550/arXiv.2604.16625
44. Zhang R., Liu G., Liu Y. et al.. (2026). Toward edge general intelligence with agentic AI and agentification: Concepts, technologies, and future directions. *IEEE Commun. Surv. Tutor.* **28**:4285–4318. DOI:10.1109/COMST.2026.3651702
45. Qu G., Chen Q., Wei W. et al.. (2025). Mobile edge intelligence for large language models: A contemporary survey. *IEEE Commun. Surv. Tutor.* **27**:3820–3860. DOI:10.1109/COMST.2025.3527641
46. Cao S., Mao Z., Gonzalez J.E. et al. (2026). K-search: LLM kernel generation via coevolving intrinsic world model. *arXiv preprint*. DOI: 10.48550/arXiv.2602.19128
47. Brooks R.A.. (1986). A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **2**:14–23. DOI:10.1109/jra.1986.1087032
48. Zheng Y., Chen Y., Qian B. et al.. (2025). A review on edge large language models: Design, execution, and applications. *ACM Comput. Surv.* **57**:1–35. DOI:10.1145/3719664
49. Yang B., He L., Ling N. et al. (2023). EdgeFM: Leveraging Foundation Model for Open-set Learning on the Edge. *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems* (*Association for Computing Machinery*), pp: 111–124. DOI: 10.1145/3625687.3625793
50. Gill S.S., Golec M., Hu J. et al.. (2025). Edge AI: A taxonomy, systematic review and future directions. *Clust. Comput.* **28**:18. DOI:10.1007/s10586-024-04686-y
51. Kang Y., Hauswald J., Gao C. et al.. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Comput. Arch. News* **45**:615–629. DOI:10.1145/3093337.3037698
52. Varghese T.G., Kochuvila S., Kumar N. et al.. (2026). Hybrid coordination framework for centralized task allocation and execution in heterogeneous multi-robot systems. *IEEE Access* **14**:61573–61595. DOI:10.1109/ACCESS.2026.3680629
53. Sun L., Yang Y., Duan Q. et al. (2025). Multi-agent coordination across diverse applications: A survey. *arXiv preprint*. DOI: 10.48550/arXiv.2502.14743
54. Alhanahnah M., Boshmaf Y. and Gehani A. (2024). SoK: Software debloating landscape and future directions. *Proceedings of the 2024 workshop on forming an ecosystem around software transformation* (*Association for Computing Machinery*), pp: 11–18. DOI: 10.1145/3689937.3695792
55. Manadhata P.K. and Wing J.M. (2011). A formal model for a system's attack surface. *Moving target defense* (*Springer*), pp: 1–28. DOI: 10.1007/978-1-4614-0977-91
56. Huang X., Ruan W., Huang W. et al.. (2024). A survey of safety and trustworthiness of large language models through the lens of verification and validation. *Artif. Intell. Rev.* **57**:175. DOI:10.1007/s10462-024-10824-0
57. Nguyen D.C., Ding M., Pathirana P.N. et al.. (2021). Federated learning for internet of things: A comprehensive survey. *IEEE Commun. Surv. Tutor.* **23**:1622–1658. DOI:10.1109/COMST.2021.3075439
58. Wu J., Liu J., Liu Y. et al. (2025). A survey on cloud-edge-terminal collaborative intelligence in AIoT networks. *arXiv preprint*. DOI: 10.48550/arXiv.2508.18803
59. Levy S.D. (2025). TinyEKF: Lightweight C/C++ Extended Kalman Filter with Python for prototyping. GitHub. Available at: https://github.com/simondlevy/TinyEKF (accessed July 5, 2026).

FUNDING AND ACKNOWLEDGMENTS

This work was supported in part by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM504, and National Natural Science Foundation of China No.62302381, No.52441602. The Authors are with the National Key Laboratory of Human-Machine Hybrid Augmented Intelligence and Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Xi'an, Shaanxi, China.

AUTHOR CONTRIBUTIONS

P.R. and G.F. were responsible for the conceptualization, study design, and drafting of the initial manuscript. S.L., Y.Z., and Q.D. conducted data collection, performed data analysis, and interpreted the results. T.X., W.Z., and N.Z. contributed to the literature review, engaged in discussions, and revised the manuscript. All authors contributed to the manuscript and approved the final version.

DECLARATION OF INTERESTS

The authors declare no competing interests.

DATA AND CODE AVAILABILITY

Data are available from the corresponding author upon reasonable request.