2025-10-2

Agentic Architectures for Robotics: Design Principles and Model Abilities

General Robotics

The goal of robotics is to create general purpose perception-action systems that can adapt to diverse tasks, environments, and embodiments. Yet, many of today's approaches are constrained by brittle pipelines and monolithic models that fail to generalize, requiring significant effort to adapt to new scenarios. Recent work on agentic workflows, where large language models (LLM) are able to plan, reason, and coordinate external tools, has introduced a distinct architectural paradigm in building software systems quickly and efficiently. We argue that such agentic architectures which can compose core skills such as perception, planning, and control are the key paradigm for general robotics. We identify three foundational design principles for an agentic architecture to apply to robotics: (1) accessible AI skills that expose diverse modular capabilities as LLM-ready tools; (2) unified robot APIs that standardize interfacing with multiple embodiments through common abstractions; and (3) LLM-driven orchestration capable of planning, code synthesis, memory, and feedback-driven adaptation. We present our instantiation of agentic robotics through these principles within the General Robot Intelligence Development (GRID) platform, and evaluate it through behavioral studies across multiple robots and tasks paired with system benchmarks. Results show that GRID enables flexible behavior that scales across embodiments and applications, positioning this approach as a foundation for transforming robotics from rigid, task-specific systems into general-purpose platforms capable of composing intelligence dynamically to meet real-world demands.

1. Introduction

The past few years have seen a surge of progress in artificial intelligence, particularly in the development of large language models (LLMs) and vision-language models (VLMs). Trained on massive corpora and enabled by the transformer architectures [1], these models demonstrate remarkable generality across natural language processing, code generation, multimodal reasoning, and interactive dialog. Beyond raw modeling capacity, an important insight has emerged: LLMs function effectively as *agents*, capable of decomposing high-level goals into structured tool invocations. Through prompting and orchestration, they can coordinate complex workflows, generate and execute plans, and interact with external systems - workflows that are already evident in domains such as software development and computer use among others. These examples suggest that the strength of LLMs lies not only in modeling ability but also in orchestrating and reasoning about heterogeneous components into coherent, adaptive workflows across use cases.

In robotics, generality has recently been pursued through large foundation models trained on robot demonstrations, with the hope that a single system could internalize diverse embodied behaviors (e.g., [2, 3, 4]). While promising, these models are costly to train, data-hungry. Unlike language or vision, where vast corpora of data are already available, robotic interaction unfolds in the physical world, spanning an effectively unbounded distribution of embodiments, environments, objects, and dynamics. Current imitation-driven approaches risk overfitting to demonstrations, tightly mimicking low level controls rather than learning task concepts. Moreover, as black box neural networks, their outputs are rarely interpretable, making it hard to diagnose and fix failures.

We argue for an alternative paradigm rooted in modularity. Rather than treating robot capabilities as monolithic systems, we propose representing them as composable modules. A picking task, for example, does not require a single model trained to imitate motor actions on every possible scenario; instead, it can be solved through coordinated use of perception to identify the object, planning to determine an approach, and control to execute the grasp in context. Modular models can be trained more effectively and at greater scale in isolation than an end-to-end model, and the resulting system gains three key advantages: interpretability, since failures can be attributed to specific modules; extensibility, because new skills can be added without retraining; and reusability, as the same modules can be applied across diverse embodiments.

However, modularity alone does not constitute intelligence. Skills—whether detectors, planners, or control primitives—are bounded functions: they provide capability, but not the judgment of when or how to apply it. Intelligence emerges in the orchestration: selecting and sequencing skills, monitoring outcomes, and adapting in response to feedback. LLM-based agents bring precisely this, with their distinctive strength lying in composition and reasoning. Rather than extending next-token prediction directly to robotics, we argue that a more effective method is to use LLMs as agents that decompose tasks, assemble workflows, and generate explicit programs integrating modular skills to solve robotics problems. Critically, this orchestration produces programs—explicit, auditable sequences of skill calls or generated code—that make behavior interpretable, extensible, and adaptive. Thus, we reframe the problem of robot autonomy as one of skill discovery, composition, and orchestration. Early modular paradigms such as the subsumption architecture [5] gestured toward this idea, but LLM-driven agents provide the missing reasoning mechanism that makes it practical.

Realizing this vision requires architectural changes tailored to robotics. Complex tasks often demand multiple models running in sequence or parallel—e.g., combining detection, pose estimation, and planning in a single loop—which exceeds the limits of typical onboard compute. Edge devices remain useful for reflexive behaviors but cannot sustain the diversity or concurrency required for composition. Cloud or remote infrastructure, by contrast, provides enhanced compute and elastic access to diverse skills, concurrent execution, and rapid switching across capabilities, making it a better choice for agentic robotics.

In this paper, we present extensions to the General Robot Intelligence Development (GRID) platform [6], an architecture for agentic robotics which instantiates these principles. Our contributions are threefold:

1. an approach for formalizing perception, planning, skill creation, and robot deployment capabilities as modular callable tools.
2. a scalable execution layer to enable agentic robotics, where skills can be hosted remotely to support diversity, elasticity, and parallel composition.
3. LLM-based agentic framework that can create, compose, and refine deployable skills, and also build and retrieve from memory.

We evaluate GRID first through system benchmarks covering area like remote inference latency, communication protocols, and deployment trade-offs to establish the viability of proposed infrastructure. We follow that with qualitative demonstrations of agentic behaviors across robot form factors and also evaluate skill creation, composition, reasoning, and finally the notion of building and using memory. Together, these results establish agentic architectures as a practical pathway toward adaptive, interpretable, and scalable robot intelligence.

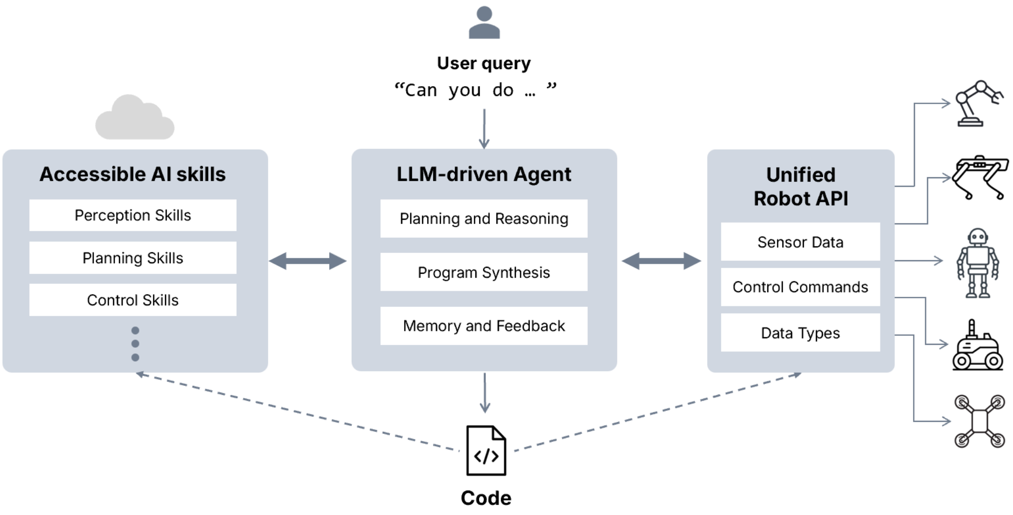


Figure 1: The GRID agentic architecture for robotics. User queries in natural language flow through an LLM-driven orchestration layer that composes AI skills and unified robot APIs to generate code for robot behaviors.

2. Architectural Principles for Agentic Robotics

Having established the motivation for agentic robotics, we now turn to a detailed discussion of the principles required to realize it in practice. The central challenge is to define an architecture for the GRID platform that can turn high-level goals into deployable and generalizable behaviors for robots. Intelligence in this setting comes not from any single model but from the ability to compose capabilities, reuse prior experience, and refine actions through error correction—directly addressing the limitations of monolithic approaches. We identify three foundational pillars that structure the GRID architecture:

1. **Skills as modular, accessible units:** AI models and robotics routines, spanning perception, planning, and control, exposed with structured inputs and outputs and hosted in scalable environments.
2. **Standardized deployment APIs:** robot actuation and sensing routines refactored into common interfaces that abstract embodiment differences and simplify integration.
3. **Composition and memory:** LLM-powered workflows that chain existing skills and synthesize new ones, augmented by multiple forms of memory to encode observations, experiences, and domain knowledge.

Together, these principles define an architecture designed to scale across tasks, embodiments, and environments, while maintaining adaptability and interpretability. In this section, we discuss the design considerations for these principles along with other supporting factors that are necessary to build an effective agentic architecture for robotics.

2.1. Skills as Modular, Accessible Units

We posit that the foundation of agentic robot intelligence is a large repertoire of atomic skills. We define a *skill* as a bounded capability that enables a robot to accomplish a meaningful function, which can be as fundamental as detecting objects, estimating depth, planning grasps, or planning a trajectory. Skills may also be hierarchical: a complex routine, such as a robot opening a door, can encapsulate multiple atomic skills, integrating, for example, semantic segmentation with downstream control. This hierarchical view allows both elementary and composite capabilities to be represented uniformly, bridging low-level functions with more abstract reasoning. Importantly, treating capabilities as modular skills also abstracts away modality: the same orchestration framework can incorporate perception from RGB cameras, LiDAR-based SLAM, or even ultrasonic sensing, provided these functions are exposed as interoperable blocks. In this formulation, skills encompass both modern neural networks and classical modules such as motion planners; the essential criterion is that they provide bounded, composable functions which, when integrated, support robust and generalizable robot behavior. In practice, modularity often unlocks generality: an object detector and a high level path planner can be equally applicable across quadrupeds, humanoids, wheeled robots, and drones.

The design choice for where and how to host these skills is equally important as the skills themselves. A common tendency in robotics is to host models directly on the robot, often on embedded GPUs. While this enables self-contained autonomy, it is fundamentally constrained: most embedded platforms can run only a few specifically optimized models, with little room for concurrency. Moreover, edge deployment often demands labor-intensive distillation, compression, and quantization, tuned to specific hardware architectures which is difficult to generalize.

In GRID, we instead favor cloud platforms and remote GPU clusters—not simply as larger accelerators, but as elastic execution environments. Skills should be hosted as auto-scalable services, where each skill can occupy a compute resource and remain ready to be invoked. This allows multiple workloads to coexist, with diverse models running concurrently and scaling independently, while the agent orchestrates them at runtime. The result is not just faster inference but a fundamentally broader skill set that can be composed and reused across embodiments.

We also build the skills in such a way that they expose structured inputs and outputs rather than ad hoc interfaces, making interoperability between skills possible which is essential for composition. Standardized APIs expose these skills uniformly to the agent, so that orchestration can remains consistent regardless of the robot or its deployment environment.

A central advantage of this modular and service-oriented design is reusability. Once defined, skills can be invoked across tasks and heterogeneous robots, and can also support coordinated multi-robot fleets that share a common library of capabilities. Platform-specific routines, such as locomotion policies, remain naturally bound to the edge for safety, but higher-level perception, planning, and reasoning skills become portable services that unify diverse platforms under a shared infrastructure. In practice, this means not only that a rapid experimentation is more feasible, especially in agentic contexts, but also that any new skills can be deployed and propagated fleet-wide with a single update.

2.2. Deployment Routines as Tool-Like APIs

While skills provide the computational core of perception, planning, and reasoning, we also need to access robots' own control and sensing routines for embodied execution. This layer today is highly fragmented: commercial platforms ship with heterogeneous SDKs, middleware stacks, and messaging protocols, each essential for operating a given robot but difficult to align across embodiments. From the perspective of an agent, navigating such deeply nested APIs is cumbersome, and orchestration logic becomes hard to generalize. Although LLMs have made significant advances as code-generation agents, creating clear abstractions remains optimal [7]: it reduces complexity, minimizes token overhead, and makes reasoning more reliable.

In GRID, we refactor deployment routines into more standardized functions. Instead of presenting raw SDKs to the agent on a per-robot basis, we introduce a unified API layer that abstracts capabilities at the level of robot form factors rather than individual devices. Each form factor—arm, quadruped, humanoid, wheeled, and drone—is represented by a canonical set of primitives in pure Python. For example, arms would implement functions like `robot.grasp(pose)`, and mobile robots would contain methods such as `robot.move_to(pose)` or `rotate(orientation)`. These primitives are consistent across manufacturers: an agent invoking `move_to(pose)` sees the same interface whether operating a UR5 arm or a Franka arm, while `get_image()` behaves identically whether backed by a Zed camera or a Realsense camera. Vendor-specific drivers, kinematic and dynamics details are encapsulated beneath these abstractions, so that the agent interacts only with stable, semantically meaningful primitives. Beyond the methods that implement robot functions, we also define custom types for standard robotics data elements such as positions, orientations, images, point clouds etc. This allows pipelines to be composed quickly without having to micromanage different types of arrays or encodings.

This framing accrues three benefits. First, it allows agents to reason over heterogeneous robots with a unified vocabulary of actions, enabling intra–form factor transfer and even limited cross–form factor recomposition. Second, it supports simulation-to-reality transfer: the same APIs used in simulators are directly executable on physical platforms. Third, From a programming-languages perspective, the standard types act as semantic contracts that guarantee interoperability between modules.

Finally, for robot APIs to couple efficiently with the AI skills, APIs must be paired with fast, reliable communication channels. Camera streams and sensor data must arrive quickly enough to drive perception and planning, while actuation commands must be acknowledged within strict safety margins. In Section 4, we quantify these requirements across protocols, but the design principle is clear: deployment APIs when standardized and paired with robust communication, extend a high degree to accessibility to robot embodiments themselves. This uniform interface not only allows the agent to reason consistently across diverse robots, but also directly supports the execution mechanism outlined in the previous section: skills hosted remotely on elastic infrastructure. By decoupling robot control from hardware-specific SDKs, agents can seamlessly integrate skills with embodied action, ensuring that perception, planning, and control are invoked as consistent, callable services across both simulation and real-world robots, thus unlocking generalization.

2.3. Agentic Composition

At the core of our architecture lies *agentic composition*, the layer that synthesizes perception, planning, and control into coherent behavior. Rather than producing control signals directly from sensor data, large language models structure tasks as sequences of modular skills. This separation parallels human cognition, where sensory and motor cortices operate independently, and purposeful action emerges only through their coordination. In robotics, agentic composition provides this coordinating function, converting raw capabilities into adaptive, goal-directed behavior.

With modular and accessible skills spanning a wide spectrum of robotic capabilities (2.1) and unified deployment APIs for heterogeneous robots (2.2), the next step is to make these capabilities accessible to an agent. In the context of large language models, this accessibility is provided through tools: structured functions with defined inputs, outputs, and documentation that an LLM can call when reasoning about how to solve a task.

In GRID, both AI skills (e.g. object detection, visual question answering, grasp prediction) and robot primitives (e.g. `get_image()`, `move_by_velocity()`) are made available as such tools. By exposing them through schemas and usage examples, an LLM agent can reliably discover, invoke, and compose them into workflows. This design allows LLMs to remain in the domain of reasoning and orchestration, while delegating complex subtasks to domain-expert models.

To make diverse capabilities discoverable and invocable by the agent, we adopt the Model Context Protocol (MCP), an open standard for exposing capabilities as tools with structured schemas [8]. Each skill—whether a perception model, planning algorithm, or robot API—is wrapped into an MCP server that provides:

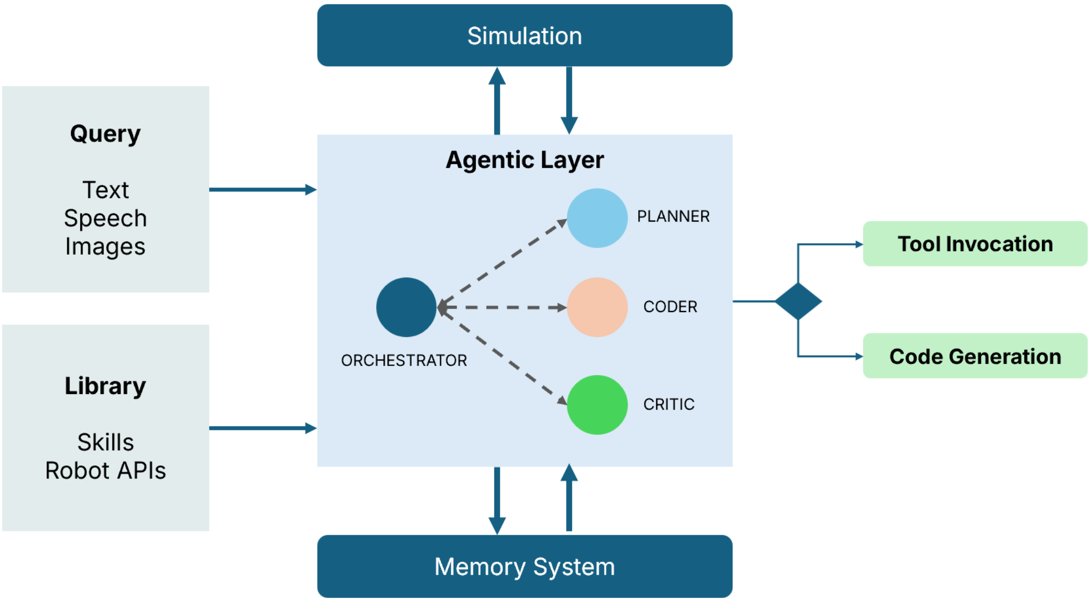


Figure 2: GRID's orchestration can utilize multiple agents assigned to specific roles to achieve complex tasks and manage context efficiently. Agents operate either in tool invocation mode for well defined problems, or code generation mode for complex tasks.

1. Tool definitions with typed parameters and return values,
2. Descriptive documentation for LLM comprehension,
3. Usage examples for quick adaptation.

Having established how skills and deployment APIs are represented, we now turn to the central challenge: how LLMs can be used to reason over these capabilities and compose them into coherent workflows. The infrastructure of callable tools and standardized deployment APIs is sufficiently expressive that robotics can, in many cases, be reframed as a program synthesis problem.

In GRID, agentic composition unfolds through two complementary modes, each suited to different classes of problems:

Tool Invocation Mode. The agent solves tasks by invoking MCP-exposed tools directly, either sequentially or in parallel, and parsing their outputs to guide subsequent calls. This supports closed-loop operation where each tool's result shapes the next decision. It is most effective when problems can be expressed as a chain of well-defined skills (e.g., detect objects → choose based on criteria → grasp).

Code Generation Mode. For problems that cannot be captured as simple tool chains, the agent synthesizes Python programs that integrate tools with custom logic. This mode is essential when mathematical reasoning, state management, or novel control flows are required. The generated code still imports MCP-exposed tools, but embeds new algorithms around them (e.g., combining monocular depth estimation with a planning routine for obstacle avoidance).

The distinction is crucial. In conventional software domains, most functionality already exists as stable APIs, which can make invocation alone sufficient. Robotics, however, confronts an open-ended diversity of tasks, embodiments, and environments, where well-defined interfaces and skill variations are rare. Code generation is therefore not an add-on but a necessity: it allows agents to construct new routines, create glue logic between heterogeneous modules, and extend the system's repertoire beyond its initial toolset.

Different stages of the composition process such as interpreting user intent, planning action sequences, writing syntactically correct code, or validating safety constraints—place very different demands on reasoning. This also requires careful prompting: instructions must be schema-constrained, grounded in retrieved documentation, and tuned for actionability, ensuring that the outputs remain executable rather than drifting into free-form language or hallucinations. While a single LLM can, in principle, handle all of these, distributing responsibilities across specialized agents often improves robustness.

Our stack is designed to support such role separation when needed, but does not require it universally. Some tasks can be solved by a single orchestrator, while more complex problems benefit from specialized roles. In our implementation, an *orchestrator agent* maintains context and handles user interaction while routing tasks to sub-agents. A *planning agent* produces structured plans with citations, a *code generation agent* translates plans into executable routines, and a *critic agent* validates outputs and proposes retries. These roles can be activated or omitted depending on task complexity.

Finally, reliability is ensured through the use of structured outputs. All agent responses are constrained to typed schemas, which enforce syntactic validity and reduce ambiguity in execution. Within these schemas, citations play a central role: each planned action must reference the documentation or examples from which it was derived. Together, schema constraints and provenance requirements establish end-to-end traceability: from a natural-language goal, to a validated plan, to generated code invoking specific tools, and ultimately to execution logs aligned with the same identifiers.

2.4. Simulation as a Sandbox for Agents

While perception/planning skills, deployment APIs, and composition form the functional pillars of agentic robotics, they do not by themselves guarantee safe or scalable development. Robotic behaviors need to be created and validated under conditions where errors are safe and iteration is rapid. Simulation provides this ability, enabling agents to explore, prototype, and refine behaviors before deployment. Observability in simulation also strengthens safety: errors and edge cases can be surfaced and audited before deployment to physical robots.

In GRID, simulation is integrated as a first-class component of the stack. Agents are able to launch simulations, select environments, execute routines, and obtain outcomes with relatively little overhead. Consistent with the design of real robot APIs, simulated robots are exposed in an equivalent manner, facilitating transfer of routines across domains and supporting large-scale experimentation. Simulation APIs are also represented as tools, permitting agents to configure environments, reset state, and query status. This enables simulation to serve not only as a testing ground but also as a source of structured feedback, allowing agents to iteratively refine behaviors and evaluate code in a controlled setting. When prompted appropriately, agents can both control and interrogate simulation, positioning it as a complementary environment for the safe and scalable development of robot capabilities.

Importantly, this capability has no direct analogue in conventional software domains. While software agents can be trained and tested directly in their operational environment, robotic agents must contend with the risks, costs, and variability of the physical world. This integration of simulation into the agentic loop is thus a prerequisite for scaling robotic intelligence safely and efficiently. Simulation functions not only as a validation step but as a generative sandbox for robotic capability: a place for agents to create, adapt, and accumulate new experiences that would be prohibitively dangerous or resource-intensive to acquire directly in real life.

2.5. Memory, Retrieval, and Reflection

Lastly, beyond the skills, deployment APIs, composition, and simulation, agents require a way to accumulate and reuse experience. In GRID, we build a robot memory layer that is organized along two complementary axes.

The first, *observational memory*, encodes what the robot perceives in an agent-interpretable form. Rather than archiving raw sensor data, observations are stored in a vector database that fuses semantic captions with dense visual embeddings. Captions, generated by vision–language models (VLMs), capture high-level context—such as scene descriptions, salient events, or judgments like "this area looks cluttered" or "there is a safety hazard." Dense embeddings, by contrast, preserve fine-grained, object-centric and pixel-level detail necessary for similarity search, anomaly detection, and object recall. Each modality offsets the other's limitations: captions alone abstract away detail, while embeddings alone lack symbolic interpretability. Their combination yields a memory that is both semantically expressive and operationally useful. When further paired with classical robotics signals—depth maps, poses, and geometric reconstructions—this memory becomes metrically grounded, enabling agents to connect semantic recall with precise spatial reasoning and control.

The second axis, *operational memory*, encodes what the robot does and how it acts. Tool invocations, generated code, and execution traces are logged as structured records, providing transparency and traceability when tasks fail, and serving as reusable demonstrations when composing new behaviors. Beyond execution traces, operational memory can also incorporate external sources of guidance—such as manuals, safety constraints, and regulatory requirements—ensuring that the agent's decisions remain aligned with both prior experience and domain knowledge. In this way, operational memory captures not only outcomes, but also the reasoning, procedures, and constraints that shaped them, supporting reflection, comparison, and continual improvement.

Together, observational and operational memory allow agents to explain and adapt their behavior over time.

3. Implementation

Our architecture rests on three principles: modular skills, standardized APIs, and agentic composition, reinforced by simulation and memory. We realize this design in the GRID platform, which unifies AI skills, robot APIs, and an LLM-based orchestrator within a single stack. GRID supports cloud-native execution for scalability while retaining edge integration for faster low level policies and safety-critical fallbacks.

Skill Abstraction: Within the GRID platform, several perception, planning, and control models were exposed as callable tools with structured inputs and outputs. Some of the models used as atomic skills during experiments in this paper were OWLv2 [9] for object detection, Grounded SAM2 [10] for semantic segmentation, ZoeDepth [11] and MIDAS [12] for monocular depth, Moondream [13] for visual question answering, and Contact Graspnet [14] for grasp prediction. All of these skills were hosted remotely on cloud services, with the agent communicating with them via REST connection.

Along with atomic capabilities mapped to certain models, higher level capabilities can also be represented as callable skills. Specifically, we utilize DreamControl [15] as a source of skills for a humanoid robot.

Robots: In this paper, we tested the GRID platform with the UR5e and Trossen WidowX robotic arms, ModalAI Starling 2 Max aerial drone, the Unitree Go2 quadruped, and the Unitree G1 humanoid. All these robots were integrated through the standardized API layer as discussed in Section 2.2. Low-level commands exposed by manufacturers were wrapped into tool-like primitives according to the form factor. These include joint control, waypoint navigation, grasping, and camera access. By exposing actuation through uniform APIs, the orchestrator can treat heterogeneous embodiments as part of the same tool library, enabling transfer of skills across robots. While these were the robots that directly supported the experiments in the paper, we note that GRID is not restricted to these choices and flexibly supports more robots.

Agent Layer: The orchestration layer is centered on an LLM agent that receives task instructions, queries available tools, and generates executable programs. In our experiments, we primarily used GPT 4.1 and GPT 5 models as the agent, while noting that the integration is LLM-agnostic, allowing other model choices to be feasible. Depending on task context, the agent may produce a sequence of Python tool calls or synthesize Python programs that bind multiple skills and robot APIs together.

For observational memory, we construct two types of embeddings: embeddings computed on top of captions such as outputs of vision language models, and object-centric image embeddings [16, 17, 18]. We fuse these embeddings into vector databases which the agent can access and retrieve records using similarity metrics.

For skill prototyping and evaluation, we use the AirGen simulator [19].

4. Results and Evaluation

We evaluate the proposed architecture along two complementary dimensions: (i) **infrastructure performance**, which quantifies the feasibility of remote/elastic skill hosting and deployment protocols in sustaining agentic workflows at scale; and (ii) **agentic behaviors**, which illustrate how the system composes, adapts, and generates robotic skills across diverse embodiments.

4.1. Infrastructure as Enabler of Agentic Workflows

The viability of agentic workflows rests on whether atomic skills can be invoked quickly and flexibly enough to sustain composition. Unlike monolithic models, which execute as a single forward pass, agents must repeatedly call into heterogeneous models, serialize intermediate results, and deliver outputs to robots within reasonable time frames. We therefore evaluated our stack along three axes: (i) inference latency across GPUs and protocols, (ii) communication overhead in robot deployment across different protocols, and (iii) feasibility of execution on edge versus cloud hardware.

4.1.1. Inference Envelope Across GPUs

We first measured the execution time for some representative models that could be commonly used in robotics across several domains—OWLv2 (object detection), ZoeDepth (monocular depth), OneFormer (panoptic segmentation), and Moondream (vision language). Each was deployed on three GPU classes (NVIDIA T4, L4, H100) to cover a spectrum of inference accelerators. We logged the backend inference latency, and end-to-end round-trip time (RTT), which includes serialization and transport overhead. Transport was tested under REST (stateless request/response) and web sockets (persistent connection), and we report the results in Table 1.

These measurements define the *skill invocation envelope*: the latency bounds within which individual skills can be called during agentic execution. Results show that inference performance scales with GPU capacity, and that persistent connections substantially reduce overhead relative to stateless HTTP. On H100 GPUs, for instance, models for depth estimation and segmentation approach 10 FPS which can sufficiently drive continuous high level navigation, while vision–language models, though slower, remain well suited for less frequent, higher-level reasoning.

Table 1: Inference and transport performance across GPUs, models, and protocols - recorded as round trip time (RTT), backend (inference) time, and network time. Values are average latencies in ms, measured from a PC in Seattle, WA to cloud VMs in the us-west1 region. All tests involved a 640x480 JPG input image.

GPU	**Model**	**HTTP**			**WebSocket**		
		RTT (ms)	Backend (ms)	Network (ms)	RTT (ms)	Backend (ms)	Network (ms)
H100	ZoeDepth	172.1	57.4	114.7	135.8	57.3	77.8
	OWLv2	230.0	183.0	46.8	215.8	186.7	28.7
	Moondream	1399.9	1351.9	48.0	1379.5	1337.0	51.7
	OneFormer	120.1	72.9	47.2	99.8	73.1	26.6
L4	ZoeDepth	224.4	100.1	124.3	183.9	98.6	85.3
	OWLv2	522.5	475.6	47.1	510.0	470.2	39.6
	Moondream	3233.9	3187.1	45.9	3223.0	3174.0	48.7
	OneFormer	202.8	154.6	48.5	183.9	154.7	29.0
T4	ZoeDepth	289.9	165.5	124.9	258.0	164.0	94.0
	OWLv2	701.3	651.1	49.0	683.0	632.0	46.8
	Moondream	3507.0	3456.0	51.7	3300	3249	50.7
	OneFormer	281.0	231.0	49.0	275.0	248.0	35.0

Table 2: One-way control / joint command latency (ms) across LAN and WAN (from Seattle, WA to US West and East cloud VMs). "adverse" indicates simulated jitter and packet loss. Bold values = best per test.

Protocol	**LAN**	**LAN (adverse)**	**US West**	**US West (adverse)**	**US East**
REST	2.552	7.722	8.388	23.827	74.752
WebSocket	1.374	3.644	4.881	15.117	36.291
gRPC	1.437	3.094	6.254	**11.271**	36.118
ZeroMQ	**1.178**	2.510	3.767	12.183	38.894
Zenoh-peer	1.262	**2.702**	**3.659**	11.927	**35.929**
Zenoh-router	1.286	2.898	3.811	12.463	36.012

4.1.2. *Communication for Robot Deployment*

Beyond inference, sustaining agentic workflows requires reliable channels for transmitting both high-bandwidth sensor data and low-bandwidth control commands. Several protocols are widely used in robotics and distributed systems: REST (stateless request–response), WebSockets (persistent bidirectional channels), gRPC (typed streaming over HTTP/2) [20], ZeroMQ (lightweight messaging without discovery) [21], Zenoh (low-latency pub/sub in peer or router mode) [22], and WebRTC (a mature standard for video transport).

We tested these protocols under several conditions—across LAN and WAN as well as adverse network conditions such as jitter (10ms) and packet loss (1%). We use two types of data: one an array of six floats to emulate control signals, and another JPEG encoded images of varying resolutions. For control data traffic, REST consistently showed the highest latencies. Persistent protocols performed much better, with LAN latencies near 1 ms and wide-area values in the 10–40 ms range. Zenoh, in both peer and router modes, tended to achieve the lowest latencies across conditions, while WebSockets and gRPC provided competitive and predictable performance. In adverse settings—both on LAN and WAN—we observed latency inflation of roughly 2–3×, underscoring the sensitivity of all protocols to jitter and packet loss, though Zenoh and gRPC degraded more gracefully than REST or WebSockets. Finally, the results also the importance of geolocating the elastic compute resources. We show these results in Table 2.

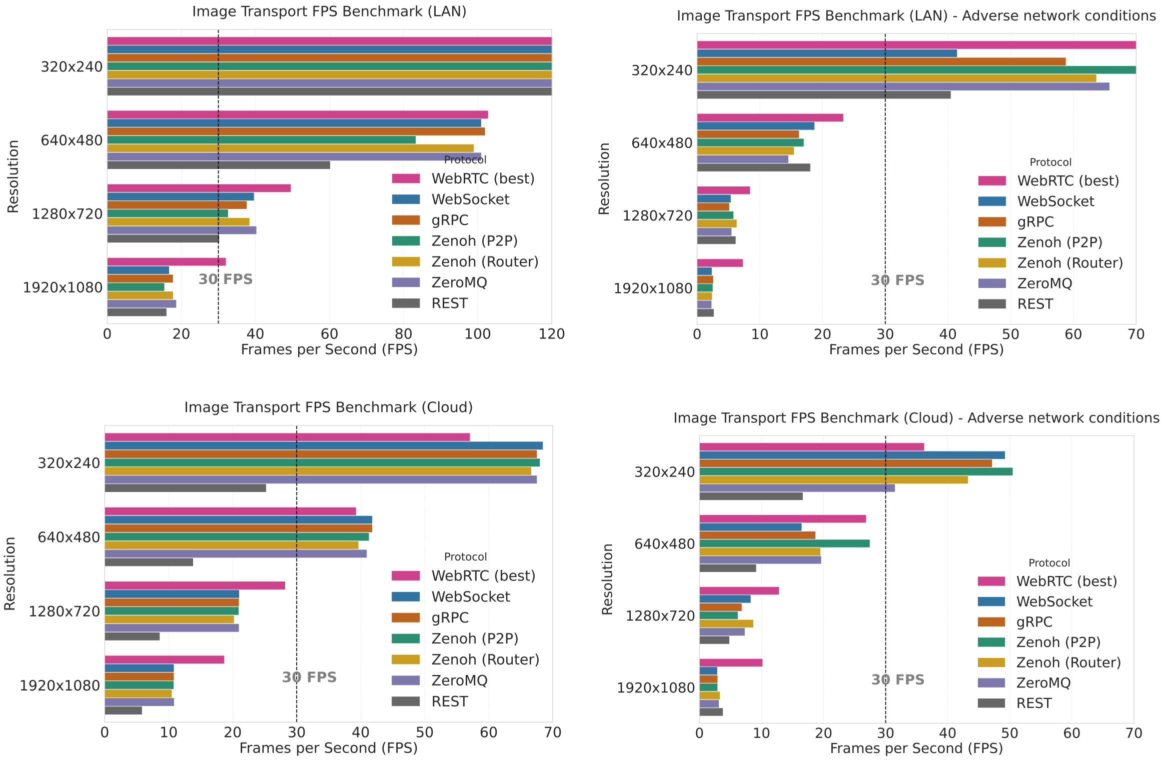


Figure 3: Image transport latency comparison across protocols. Benchmark conducted over LAN and WAN (Seattle, WA to cloud VM in US West). Adverse network conditions indicate simulated jitter and packet loss.

Comparing protocols, WebSockets show competitive latency but lack scalability, limiting their role in distributed settings. ZeroMQ offers the lowest raw LAN latency but provides only minimal messaging primitives without discovery or routing. gRPC performs reliably, especially under adverse WAN conditions, though its request–response model is less suited to streaming. Zenoh provides the best balance, combining low latency with a pub–sub abstraction well aligned with multi-robot workloads.

For image streaming, we evaluate across a range of resolutions: QVGA to Full HD and show the results in Figure 3. In image transport, REST proved impractical, and WebSockets performed well at low resolutions but scaled poorly with larger frames. Zenoh maintained low latencies in peer-to-peer mode, with modest router overhead. WebRTC, though incurring encode/decode costs, provided the most efficient bandwidth use, and at higher resolutions it outperformed the other protocols in effective end-to-end latency. Its practical benefits—such as hardware codec support (H.264/H.265/VP9/AV1), fine-grained rate control, and resilience in complex network topologies—make it particularly attractive for vision workloads.

These results suggest that Zenoh can be a strong choice for control channels, providing consistent low-latency operation, while WebRTC is well suited for high-resolution vision workloads, balancing compression, scalability, and deployment convenience. Crucially, these protocols are complementary rather than mutually exclusive: a practical agentic system can use Zenoh for control and WebRTC for perception, aligning each channel with the strengths of its transport.

It is important to note that the absolute numbers reported here are highly dependent on experimental conditions, including hardware, network topology, and geographic location. The key contribution is therefore comparative: relative performance trends across protocols, rather than fixed benchmarks, that guides the design of agentic robotic systems.

4.1.3. Edge vs. Cloud

Our experiments underscore the qualitative limits of edge execution for agentic workflows. Sustaining parallel execution of AI models is difficult for these computers, and even modest compositions can push memory and latency constraints on embedded devices.

In contrast, remote execution readily supports the diversity and concurrency required for agentic workflows. As a test, we ran multiple perception models in parallel using elastic cloud GPUs. We started with a monocular depth model running atop RGB images which operated at ∼5 Hz; adding semantic segmentation in parallel (effectively akin to a robot constructing a metric semantic map) still kept the rate at 5 Hz, as both inferences could run on separate resources. Adding an object detector reduced its stream to ∼3 Hz, and a vision–language model to ∼1 Hz. In this implementation, coroutines were awaited as a group (using `asyncio.gather` in Python), so the pipeline would be gated by the slowest task. Switching to a multi-threaded design removed this limitation, with each model running at its native rate. More broadly, this demonstrates a key principle for agentic workflows: non-blocking execution allows fast skills to remain fast while slower ones contribute opportunistically.

These results highlight that elastic execution, where diverse skills can be run in parallel, swapped in and out, and scaled without hardware reconfiguration provides the foundation for agents to deploy complex capabilities in real-world scenarios.

4.2. Agentic Behaviors

With this foundation in place, we turn to qualitative evaluations of the agentic capabilities of GRID. The central hypothesis of this work is that an agent equipped with modular, effective skills and a unified robot API can flexibly solve tasks across domains.

To test this, we evaluated the architecture across multiple robot form factors and task categories, spanning manipulation, navigation, reasoning among others. In each case, the agent was given access to a library of perception, planning, and control models, and deployment APIs for the corresponding robot. The results show that GRID is not only able to orchestrate existing skills, but also create new ones, adapt previously learnt skills to new use cases, and solve reasoning-intensive problems that are beyond the reach of current monolithic robot foundation models.

4.2.1. Cross-Domain Skill Composition

We first examine whether GRID could flexibly combine perception and control skills across distinct robot embodiments. Four representative cases illustrate the generality of this capability.

Robot Arm Manipulation. For a UR5e arm with a Zed stereo camera, we tasked GRID with composing a 'pick-and-place' skill in code-synthesis mode. The agent was able to achieve reasonable behavior in a single shot, by chaining together a segmentation model to find the object, a grasp skill powered by a two-finger grasp detector to determine how to pick it, and a vision-language model to identify the placement location. We do note that the success of such a composition is heavily dependent on the quality of the individual skills themselves: for example, the grasping skill was not merely the pose output from the raw model itself but it was a combination of multiple poses: pre-grasp or approach pose, execution of the actual grasp itself, and then a 'lift' to raise the object. Within the scope of the agentic framework, it becomes trivial to write

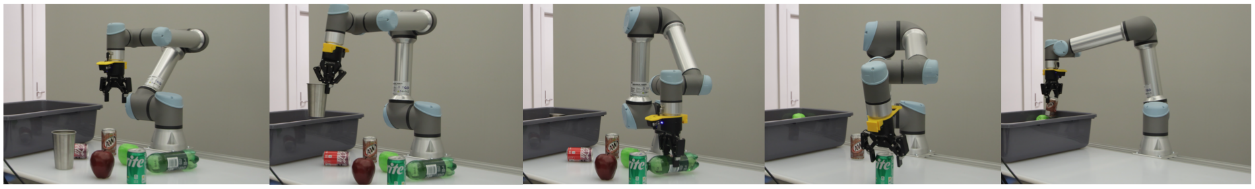


Figure 4: GRID responds to a user query 'clean up the table' by composing several skills. Modular skills allow generalizing zero-shot to various object classes.

sub-skills such as this through simple natural language instructions in a way that is particularly relevant for a user or environment, which greatly accelerates solution development.

USER: Can you use the UR5e arm to pick up the red apple and place it in the silver cup?

GRID# Approach: Initialize the UR5e arm, Robotiq gripper, and the ZED camera to detect, grasp, and place the apple.

```
rgb_image = ur5e_0.getImage(”zed”, image_type=”rgb”)
depth_image = ur5e_0.getImage(”zed”, image_type=”depth_meters”)
# Segment the red apple
seg_result = client.run(model_id=”gsam2”, image_input=rgb_image.data, prompt=”red apple.”)
...
camera_matrix = camera.getCameraMatrix()
grasps = get_grasp(rgb_image.data, depth_image.data, seg_result, camera_matrix=camera_matrix)

ur5e_0.moveToPose(pre_grasp_pose.position, pre_grasp_pose.orientation)
ur5e_0.grasp()

...
```

Highlighting the generality of modular skills such as this, the resulting composition successfully manipulated several household items like cans and fruit, as well as niche industrial objects like hard drives, without any fine-tuning.

Given that the above test was conducted in code synthesis mode, the execution of the entire skill was open-loop and hence the agent was not involved within the skill sequence. As discussed in section 2, the reasoning capability of agents shines even more when we close the loop and unlock adaptive behaviors. To test this, we registered the previously synthesized pick and place skill as a callable tool. Given a more complex command, “clean up the table”, GRID used a VLM to identify all objects, then iteratively called the 'pick-and-place' tool, and also retrying upon any failed detections/grasps until the task was complete, showcasing hierarchical composition and closed loop behavior.

USER: Please clean up the table, put everything in the bin on the left.

GRID# Thinking...

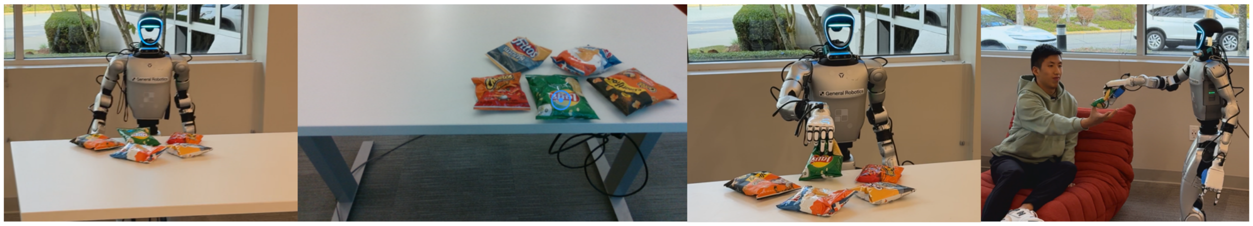


Figure 5: GRID orchestrates a humanoid multi-step task. When asked to "bring the Lays chips", it composes a program to identify the chips, pick them up, and detect and navigate to the nearest person then hand it off.

```
scene = agent.run_tool(”get_description”, prompt=”Describe the scene in front of the robot.”)
# -> "There is a table with several objects including a plastic bag, a can, and a bottle.”
```

GRID# Thinking...

```
agent.run_tool(”pick”, object=”can on the table”)
agent.run_tool(”place_object_at_named_position”, location=”left_bin”)
```

GRID# Thinking...

```
agent.run_tool(”pick”, object=”bottle on the table”)
agent.run_tool(”place_object_at_named_position”, location=”left_bin”)
...
```

Humanoid Long Horizon Task. With a Unitree G1 humanoid robot, we evaluated the integration of pre-trained skill policies within a longer sequence. Given the instruction ‘"pick up the Lays chips bag and hand it to the nearest person,"‘ GRID orchestrated a multi-stage plan. First, it used a VLM to associate the phrase "Lays chips" to a specific point in its camera image. It then invoked a pre-trained pick policy (trained using [15]) to execute the complex manipulation for the pick given tthe corresponding 3D point backprojected from its onboard camera. Finally, it used a person detector and navigation skills to rotate and search for the nearest person and walk towards them, performing a hand-off gesture (Figure 5) at the end. This shows that specialized policies can be seamlessly integrated as callable skills within a broader, agent-orchestrated workflow. In our actual experiment while executing this loop, owing to the fact that the G1 robot does not have an effective emergency stop mechanism, we manually approved each step to ensure safety while switching skills.

Quadruped Object Goal Navigation. Using a Unitree Go2 quadruped, we issued the command ‘"search the area until you find a yellow warning sign"‘. GRID synthesized a loop that composed two AI skills. The first was a prebuilt **reactive navigation** skill, which took the robot's forward-facing RGB camera image, passed it to a cloud-hosted depth estimation model, and used the resulting depth map to calculate the safest immediate direction of travel. The second was an object detector. The Go2 robot was controlled using a velocity control API that would receive a (vx, vy) command at every step. The agent-generated code called these two skills iteratively—move safely, then look for the sign—until the task's goal was met, again demonstrating the composition of multiple policies.

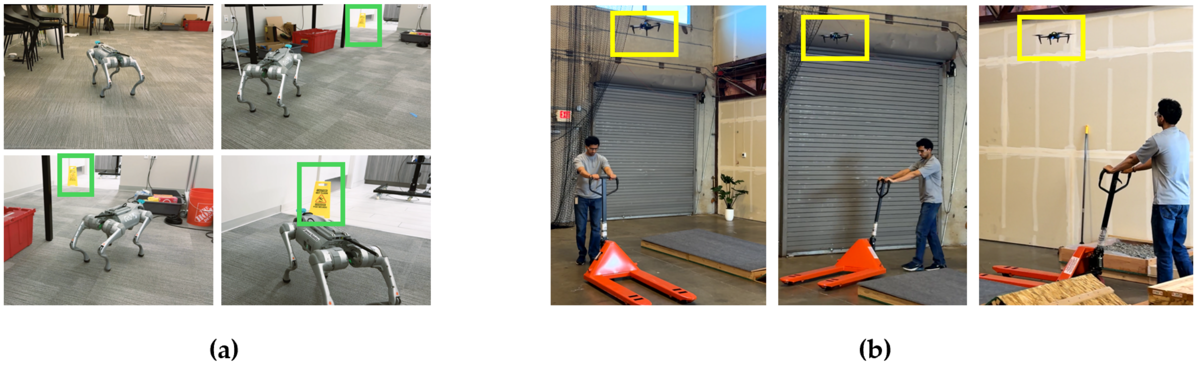


Figure 6: Modularity and accessibility of skills and robot APIs unlocks generalization across robot form factors. Results from agentically composed skills for a (a) Unitree Go2 quadruped to navigate until it finds a 'yellow warning sign' (in green). (b) ModalAI drone to follow a moving target 'orange palletjack' (drone in yellow).

Aerial Visual Landing. For an aerial drone, we tested GRID's ability to compose a skill for target tracking. We used a ModalAI Starling 2 Max drone for the experiment and expose APIs for velocity control and camera image access to the agent.Given a prompt "detect and track the orange palletjack", GRID was able to create a pipeline orchestrating two relevant skills - one a VLM to 'point' to the object of interest [13], and another a function to track the velocity of a point in pixel space, resulting in the drone moving with an object of interest to follow it from a fixed altitude.

Overall, this shows that by unifying heterogeneous robots under a standard API layer, agents can work seamlessly across form factors, applying their reasoning and composition capabilities to solve hard problems.

4.2.2. Adaptation of Existing Skills

Another test of generality is whether agents can flexibly adapt skills to new purposes. We evaluated this by asking GRID to repurpose a pick-and-place pipeline originally developed for relocation into object sorting and object scanning. Traditional pipelines are brittle: perception models are hard-wired to control primitives and often require re-engineering or more demonstrations for new tasks. In contrast, in this architecture, GRID reused its segmentation, depth estimation, and grasp prediction tools and reconfigured outputs via the control API. In one case, the UR5e was instructed to "sort Coke cans into one bins and others into another," and in another to "image this object from multiple viewpoints to scan it." To solve the latter, the agent

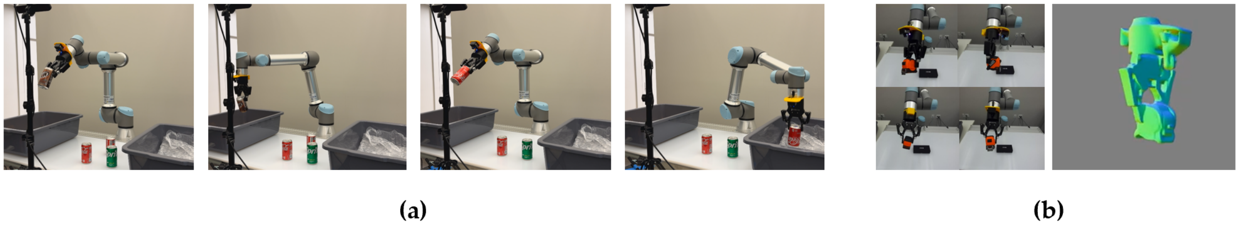


Figure 7: GRID can adapt previously built skills such as pick-and-place to new tasks. (a) When asked to sort objects by category, GRID combines visual question answering with the pick and place skill it had built. (b) When asked to image an object for 3D scanning, GRID wrote functions to bring the object to the camera and to rotate the gripper 360 degrees.

implemented a skill to bring the object up to the camera with the arm and rotate it while capturing images. These results illustrate that agentic recomposition enables skills to be flexibly adapted to new contexts without retraining.

4.2.3. Creation of New Atomic Skills

Beyond composing existing tools, we tested GRID's ability to create new, somewhat complex skills through autonomous prototyping. We tasked it with generating a general-purpose 'safe navigation' skill for a wheeled robot using only an RGB camera feed as input. The agent was given access to a library of AI models (including several RGB-to-depth networks) and a simulator.

GRID automatically initiated the simulator with the right configuration (wheeled robot), and then wrote code to pipe the RGB images through a monocular depth model MIDAS [12], and used the output to implement a reactive obstacle-avoidance routine. As part of its code generation process, the agent also implemented several logging statements and hooks to monitor feedback variables in the code which it would later parse to understand whether the skill was successful.

GRID then automatically executed and evaluated its own code within the simulator, iteratively refining the logic until the behavior satisfied the original constraints (no collisions for 60 seconds). More details on the results of this process can be seen in Appendix B.2.

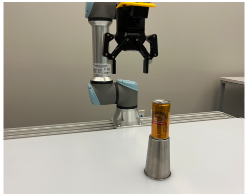
We highlight that an ability of this kind where an agentic framework can automatically start up simulations, parallelize across different environments/conditions, and iterate on skill writing can be a highly effective paradigm for creating or adapting robotic behaviors.

4.2.4. Reasoning Capabilities

While many of our experiments focused on perceptual or compositional capabilities, we also examined whether the architecture could support reasoning-driven behavior. Reasoning is a natural strength of large language models, and in an agentic robotics stack it can be directly applied to questions of task feasibility, constraint satisfaction, and symbolic planning.

We began with a simple feasibility test: in an environment where a soda can was placed on top of a cup, GRID was asked to retrieve the cup using a UR5 arm. GRID queried a VLM to understand the scene characteristics first, which reported that the cup had a soda can resting on top of it. Rather than attempting to grasp the object directly, GRID generated a stepwise plan: remove the soda can, place it aside, and only then pick up the cup. This reasoning chain mirrors the kind of logical deduction LLMs routinely perform in text-based domains, but here it was grounded in embodied execution through perception APIs and control skills. The result demonstrates that feasibility reasoning, often left implicit in robotic systems, can be surfaced explicitly and translated into safe, interpretable behavior. Unlike a policy that might attempt a direct grasp and risk failure or damage, GRID's explicit reasoning avoided this hazard, underscoring the safety benefits of interpretable intermediate plans.

USER: Can you use the UR5e arm to pick up the silver cup?



GRID# 1 Scene Analysis:
- The silver cup is underneath a soda can on the table.

2 Move Obstructing Object:
- Pick up the soda can that is on top of the silver cup.
- Place the soda can in a temporary location on the table to clear the way for the silver cup.

3 Pick Up Silver Cup:
- Once the soda can is moved, pick up the silver cup from the table.

4 Completion:
- Confirm that the silver cup is securely in the gripper.

At the other extreme, we evaluated whether GRID could integrate complex symbolic reasoning with embodied manipulation by tasking GRID to have a robot arm play chess. We used the Trossen WidowX robot arm and added skills for board state detection and for piece localization. We initially used GPT-5 as the agent who would receive a representation of the game state and generate legal moves in response, and also issued appropriate control commands for the arm to reposition pieces on the board. This task lies far outside the distribution of current monolithic vision–language–action models, which cannot perform such reasoning-intensive, out-of-domain tasks without significant retraining involving immense manual effort. In contrast, the agentic stack leveraged the reasoning capacity of LLMs, coupled with perception and control, to carry out embodied symbolic problem-solving in a modular way.

USER: Can you tell me which opening we just played?

GRID# The opening we've played is the Queen's Gambit Declined, specifically after the moves: 1. d4 Nf6 2. c4 e6 3. Nc3 d5 In the Queen's Gambit Declined (QGD), White offers a pawn with 2. c4 to gain central control, while Black declines the gambit by supporting their d5 pawn with 2...e6 rather than capturing the c4 pawn. This opening can lead to solid positions for Black with good central control and a strong pawn structure....

In reasoning-intensive tasks such as chess, we observed that GPT-5 outperformed GPT-4.1 in overall play quality, though neither model was tuned for competitive performance. To further probe the flexibility of the architecture, we integrated Stockfish as an additional move generator skill, with the LLM retained in the orchestration role. This hybrid setup markedly improved quality of play: Stockfish supplied tactical move choices, while the LLM maintained the game state, controlled the robot, generated explanations, and mediated user interactions. For example, when asked mid-game which opening had just been played, GRID produced an appropriate description and explanation in context.

The purpose of this experiment is not to benchmark chess models, but to demonstrate that domain-specific systems can be incorporated into the agentic workflow as callable skills. By allowing specialized tools to be combined with general reasoning, the architecture supports flexible integration of external capabilities, whether in symbolic domains such as games or in robotic tasks requiring specialized planning, perception, or control.

These experiments show that reasoning, often considered an abstract strength of LLMs, becomes directly applicable in robotics when embedded in an agentic architecture. From assessing feasibility in everyday manipulation to executing symbolic games like chess, the same underlying reasoning machinery was grounded in perception and action, underscoring its importance for safe and adaptable behavior in open-ended tasks.

4.2.5. Situational Awareness Through Observational Memory

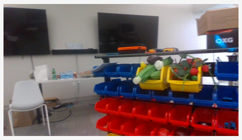
A key feature of the agentic architecture is the ability to augment real-time perception with accumulated experience. To evaluate this capability, we constructed an observational memory by running a mobile robot through an office environment and logging embeddings of its visual stream into a vector database.

USER: I am looking for a box that says OXG, where is that?

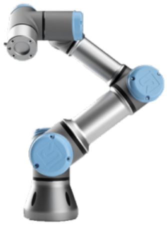
GRID# The box labeled "OXG" was observed on a black shelf unit in a room that appears to be a workspace or office. The shelf contained a variety of colorful plastic bins filled with fruits and vegetables. So, if you are looking for the "OXG" box, head to the area with the black shelving unit that holds bins of produce.

USER: Can you show me where you saw it?

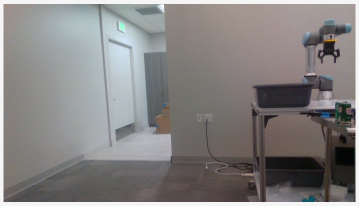
GRID#



USER: Have you seen this object?



GRID# Here is the closest matching object I have seen.



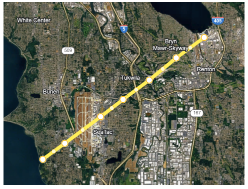
We evaluated this memory system through a set of qualitative tasks. First, GRID was able to answer retrospective queries such as “Where did you see this particular object?” or “Was the trash bin full?” by retrieving relevant frames via caption embeddings and grounding them in the robot’s exploration map. Second, the system enabled direct image recall: when asked to “show me where you saw this particular box,” GRID surfaced the corresponding frame and contextualized its location. Third, we tested multi-modal retrieval by providing GRID with only a query image (in this case, we gave it a stock image of the UR5 arm) and asking whether it has seen something like that. GRID matched the query image against stored frames and returned the most relevant results. We note that the UR5 arm was not closely imaged as part of the robot trajectory, which speaks to the importance of image embeddings. While textual captions would have missed the detail required to identify a faraway object from an image, object-centric image embeddings have an advantage in being able to encode much more granular understanding in a compressed way which is paramount for robot memory.

These experiments demonstrate that observational memory provides a complementary substrate for reasoning beyond instantaneous perception. Memory allows GRID to recall past conditions, identify anomalies, and recover references to previously seen objects. Such capabilities support practical use cases including locating misplaced items, verifying operational conditions (e.g., checking whether bins were full, or a machinery was in use). By combining semantic captions with visual descriptors, the memory enables both natural language reasoning and appearance-based search, broadening the range of queries GRID can resolve.

4.2.6. Grounding Decisions in Domain Knowledge

We conducted another experiment to highlight the possibility of using domain-specific knowledge to constrain and guide robot behavior, which falls under the category of operational memory. We used the field of drone mission planning as a test case for domain-specific knowledge, given its clear regulatory constraints, well-defined operational rules, and immediate practical relevance for embodied agents.

USER: Requesting approval for drone mission plan: (visual depiction below, actual mission involved latitude/longitude waypoints)



GRID# REQUIRES AUTHORIZATION. 14 CFR §107.41: Operation in the vicinity of airports: The mission is within 5 miles of multiple airports, which likely places it in controlled airspace (Class B, C, D, or E surface). RECOMMENDATIONS: - Obtain LAANC or written ATC authorization prior to flight if operating in controlled airspace.

To begin, we embedded the FAA Part 107 manual into GRID's operational vector database, enabling GRID to retrieve relevant sections while responding to the user. Given a mission request, GRID classified it as "go," "no go," or "valid with modifications" depending on whether the proposed flight complied with regulatory constraints. For example, when presented missions that crossed restricted airspace, the agent was able to call geocoding APIs to evaluate the locations of the waypoints, and reject them; while those exceeding altitude limits were flagged with corrective guidance. We show more examples of this test case in Appendix B.3.

Beyond aviation, this experiment highlighted a broader value proposition: any industry- or site-specific information—such as safety protocols, facility timetables, or operational guidelines—can be encoded into a retrievable memory. This allows agents to ground their decisions not only in perception and planning skills but also in the normative and contextual knowledge that governs safe and effective operation.

4.3. Comparison of Multiple Agentic Tools

To validate the architectural principles underlying GRID, we conducted an ablation study examining how different components contribute to successful robotic code generation. We evaluated three state-of-the-art code generation models—Claude Code, Codex, and GitHub Copilot—under progressively reduced support conditions, measuring their ability to generate executable robot control code. The agents were tasked with creating a script for pick-up and place using the UR5e arm. The cases considered were the following.

Case 1 – Full GRID Integration: Agents had access to GRID's unified robot APIs and cloud-hosted skills (e.g. perception models, grasp planners).
Case 2 – APIs Only: Agents retained access to GRID's unified robot APIs but lacked AI skills, forcing them to approximate perception and planning.
Case 3 – No GRID Support: Agents were forced to work directly with manufacturer SDKs, without GRID APIs or skills.

Table 3: Ablative comparison of GRID support levels. Full program outputs are in Appendix C.

Condition	Claude Code	Codex	Copilot	GRID Interface
Case 1: Full GRID (C.1)	2	1	3	1
Case 2: APIs Only (C.2)	Fail	Fail	Fail	–
Case 3: Raw SDKs (C.3)	Fail	Fail	Fail	–

With full GRID support (Case 1), all agents generated working programs, though with varying efficiency: Codex succeeded in one attempt, Claude Code required more iterations, and Copilot resolved initialization issues after manual guidance. Removing cloud-hosted skills (Case 2) led to universal failure, as agents were unable to construct functional perception or planning pipelines or retrieve any available models. Without both APIs and skills (Case 3), the output was reduced to low-level joint commands—syntactically valid but behaviorally unusable.

Three conclusions follow. First, unified robot APIs are necessary but not sufficient: they simplify control but the bulk of any perception-action loop requires readily available AI skills that are not as ubiquitous as in general software domains. Second, modular skills require orchestration—without proper composition mechanisms, agents cannot pipeline data between modules.

Most importantly, the study highlights a key distinction between software and robotics. In software, agents can often succeed by calling several APIs of existing, well established tools. In robotics, without the scaffolding that GRID provides—unified APIs, modular skills, and orchestration—agents are trapped in low-level details and fail to produce viable behaviors. GRID supplies the missing structure that transforms general-purpose code generators into effective robotic programmers. This ablation also underscores a safety advantage: without structured APIs and modular skills, agents are prone to issuing low-level unsafe commands, whereas GRID enforces safer abstractions.

4.4. Limitations

While the results presented here highlight the promise of agentic architectures for robotics, we do note some limitations.

First, agents are inherently constrained by the quality of the underlying skills they orchestrate. Perception models may hallucinate, depth estimators can produce inaccurate ranges, and planners may fail in unfamiliar configurations. Such errors propagate upward, compromising overall behavior. This underscores the importance of modularity: because skills are exposed through explicit interfaces with observable outputs, the system can incorporate validation layers and safety checks.

Second, what we present is best understood as a universal *framework* rather than a universal *solution*. The architecture generalizes across domains and embodiments, but reliable performance depends on the continued improvement of the skills themselves and on careful selection of modules suited to a given task. The framework enables this interchangeability, but it does not eliminate the need for high-quality, domain-specific capabilities.

Finally, the use of LLMs introduces its own challenges. Performance is sensitive to prompting strategies, examples, and context management. Large codebases, extensive execution histories, or heterogeneous tool catalogs can overwhelm the agent if naively presented. Effective orchestration therefore requires careful prompt design and the use of retrieval mechanisms (e.g., RAG) to surface only the most relevant records at inference time. Without these measures, context overflow and drift can degrade reasoning quality.

5. Related Work

5.1. Monolithic foundation models vs modular compositions of core AI skills

Recent advancements in artificial intelligence have seen a significant trend towards training monolithic foundation models for various applications [23]. In robotics, a widely held aspiration is to harness the scaling laws associated with transformer-based architectures [24, 25] to achieve a “ChatGPT moment”, unlocking new emergent abilities and achieving robustness.

However, unlike the natural language processing (NLP) domain, where vast amounts of text data are readily available via the web, robotics faces significant challenges in accessing similarly extensive datasets. Despite this, some research groups are actively collecting thousands of hours of manually teleoperated robot trajectories to train policies using imitation learning [26, 27, 28, 2, 3, 29, 30, 31, 32, 33, 34]. However, this is very expensive and prominent roboticists have noted that the scale of these datasets remains considerably smaller compared to that of text data, as highlighted in [35].

In this paper, we adopt a modular approach, separating various components of the perception system from aspects such as motion planning and grasp generation. These modules are often well-studied in isolation and already have associated foundation models trained with significantly more data than could be achieved solely with teleoperated robot trajectories (e.g. [11, 36, 14, 37])

While modular approaches are in some sense more traditional than the monolithic foundation models above, they continue to be popular in the community for the reasons that we’ve articulated in this paper (e.g. ease of debugging, the availability of robust off-the-shelf solutions, and the avoidance of reinventing the wheel, etc). Recent examples in both navigation [38, 39, 40, 41] and manipulation [42, 43] further underscore their continued relevance.

5.2. LLM and Agentic approaches in Robotics

Recent advances in the LLM space, particularly in tool use and code generation [44, 45, 46, 47, 48, 49, 50], have opened up new avenues for fully automating these modular pipelines, which previously required manual design. A number of works have leveraged LLMs for high-level planning in robotics [51, 52, 53, 54, 55, 56, 57, 58], either directly as policies, or by leveraging their code generation capabilities (e.g., [59, 60, 7, 58, 61, 62, 63]) or for designing reward functions used during training [64, 65, 66, 67]. Recent work has also begun to explore the role of memory in agentic systems, where agents can accumulate, retrieve, and reflect on prior experiences to guide future behavior [68].

Our work primarily builds upon approaches that directly output code. This method has historically been more challenging due to the need for manual prompt engineering and a lack of sufficiently large collections of “skills” to fully demonstrate the potential of agentic robotics. However, by combining modern LLMs better trained for code generation with MCP standardization for tool calling and a cloud-hosted library of atomic skills, alongside a unified evaluation and deployment language, we can more completely realize the vision of agentic robotics.

6. Conclusions

This work introduced an agentic architecture for robotics in the GRID platform, which views robot autonomy as the result of modular composition orchestrated by LLM agents. By exposing perception, planning, control, and deployment routines as callable tools, GRID enables agents to generate robot programs that flexibly integrate diverse skills, adapt to novel tasks, and operate across heterogeneous embodiments. Crucially, it is not only the existence of such tools but their accessibility and composability that make agentic workflows viable for robotics.

The implications are fourfold. First, agentic orchestration accelerates development and deployment in robotics, much as software agents have done in programming: new skills can be assembled, refined, and reused rapidly, lowering the barrier to creating complex behaviors and delivering value with robots. Second, simulation as a first-class tool allows agents not only to evaluate routines but also to generate and adapt new skills at scale before deployment—turning simulation into a generative engine for capability. Third, by grounding control in natural language and memory, robots can converse about their environment, recall past experiences, and situate themselves in domain-specific contexts, enabling collaborative problem solving rather than brittle, hard-coded execution. Fourth, and most critically for deployment, transparency and interpretability become built-in safety features: because intermediate outputs from modular tools are observable and inspectable, developers can trace decisions, identify errors, and enforce safeguards. This makes robotic behaviors more predictable, debuggable, and trustworthy in real-world settings.

Together, these findings suggest that intelligence in robotics does not emerge from scaling monolithic networks alone but from the integration and orchestration of diverse, reusable components. GRID demonstrates that modularity, combined with agentic reasoning, yields interpretable, extensible, and adaptive behavior—providing a foundation for the next generation of embodied intelligence. As robotics systems grow in complexity, architectures that combine modularity, agentic reasoning, and transparent safety mechanisms will be essential to ensure not only intelligence, but also reliability and trust in real-world deployment.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 1).

[2] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. "π_0: A Vision-Language-Action Flow Model for General Robot Control." In: *arXiv preprint arXiv:2410.24164* (2024) (cit. on pp. 1, 21).

[3] Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. "Gr00t n1: An open foundation model for generalist humanoid robots." In: *arXiv preprint arXiv:2503.14734* (2025) (cit. on pp. 1, 21).

[4] Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. "Rdt-1b: a diffusion foundation model for bimanual manipulation." In: *arXiv preprint arXiv:2410.07864* (2024) (cit. on p. 1).

[5] R. Brooks. "A robust layered control system for a mobile robot." In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23 (cit. on p. 2).

[6] Sai Vemprala, Shuhang Chen, Abhinav Shukla, Dinesh Narayanan, and Ashish Kapoor. *GRID: A Platform for General Robot Intelligence Development*. 2023. arXiv: `2310.00887 [cs.RO]`. URL: `https://arxiv.org/abs/2310.00887` (cit. on p. 2).

[7] Sai H Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. "Chatgpt for robotics: Design principles and model abilities." In: *Ieee Access* 12 (2024), pp. 55682–55696 (cit. on pp. 4, 21).

[8] Anthropic. *Introducing the model context protocol*. URL: `https://www.anthropic.com/news/model-context-protocol` (cit. on p. 5).

[9] Matthias Minderer, Alexey Gritsenko, and Neil Houlsby. "Scaling open-vocabulary object detection." In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 72983–73007 (cit. on p. 8).

[10] Tianhe Ren, Shilong Liu, Ailing Zeng, Jing Lin, Kunchang Li, He Cao, Jiayu Chen, Xinyu Huang, Yukang Chen, Feng Yan, Zhaoyang Zeng, Hao Zhang, Feng Li, Jie Yang, Hongyang Li, Qing Jiang, and Lei Zhang. *Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks*. 2024. arXiv: `2401.14159 [cs.CV]` (cit. on p. 8).

[11] Shariq Farooq Bhat, Reiner Birkl, Diana Wofk, Peter Wonka, and Matthias Müller. "Zoedepth: Zero-shot transfer by combining relative and metric depth." In: *arXiv preprint arXiv:2302.12288* (2023) (cit. on pp. 8, 21).

[12] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.3 (2022) (cit. on pp. 8, 16).

[13] Vikhyat Korrapati. *moondream2 (Revision 92d3d73)*. 2024. URL: `https://moondream.ai/` (cit. on pp. 8, 15).

[14] Martin Sundermeyer, Arsalan Mousavian, Rudolph Triebel, and Dieter Fox. "Contact-graspnet: Efficient 6-dof grasp generation in cluttered scenes." In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021 (cit. on pp. 8, 21).

[15] Dvij Kalaria, Sudarshan S Harithas, Pushkal Katara, Sangkyung Kwak, Sarthak Bhagat, Shankar Sastry, Srinath Sridhar, Sai Vemprala, Ashish Kapoor, and Jonathan Chung-Kuan Huang. "DreamControl: Human-Inspired Whole-Body Humanoid Control for Scene Interaction via Guided Diffusion." In: *arXiv preprint arXiv:2509.14353* (2025) (cit. on pp. 8, 14).

[16] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. "Learning transferable visual models from natural language supervision." In: *International Conference on Machine Learning*. 2021 (cit. on p. 9).

[17] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, et al. "Dinov2: Learning robust visual features without supervision." In: *arXiv preprint arXiv:2304.07193* (2023) (cit. on p. 9).

[18] Mike Ranzinger, Greg Heinrich, Jan Kautz, and Pavlo Molchanov. "AM-RADIO: Agglomerative Vision Foundation Model Reduce All Domains Into One." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024 (cit. on p. 9).

[19] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. "Airsim: High-fidelity visual and physical simulation for autonomous vehicles." In: *Field and Service Robotics: Results of the 11th International Conference*. 2018 (cit. on p. 9).

[20] gRPC Authors. *gRPC*. URL: `https://grpc.io/` (cit. on p. 10).

[21] ZeroMQ Project. *ZeroMQ: The Intelligent Transport Layer*. `https://zeromq.org/`. Accessed: [Your Access Date]. n.d. (Cit. on p. 10).

[22] The Zenoh Team. *Zenoh - Zero Overhead Network Protocol*. July 2025. URL: `https://zenoh.io/` (cit. on p. 10).

[23] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. "On the Opportunities and Risks of Foundation Models." In: *arXiv preprint arXiv:2108.07258* (2021) (cit. on p. 21).

[24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. "Scaling laws for neural language models." In: *arXiv preprint arXiv:2001.08361* (2020) (cit. on p. 21).

[25] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. "Training compute-optimal large language models." In: *arXiv preprint arXiv:2203.15556* (2022) (cit. on p. 21).

[26] Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, et al. "Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024 (cit. on p. 21).

[27] Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, et al. "Octo: An open-source generalist robot policy." In: *arXiv preprint arXiv:2405.12213* (2024) (cit. on p. 21).

[28] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. "Openvla: An open-source vision-language-action model." In: *arXiv preprint arXiv:2406.09246* (2024) (cit. on p. 21).

[29] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. "Diffusion policy: Visuomotor policy learning via action diffusion." In: *The International Journal of Robotics Research* (2023), p. 02783649241273668 (cit. on p. 21).

[30] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. "Learning fine-grained bimanual manipulation with low-cost hardware." In: *arXiv preprint arXiv:2304.13705* (2023) (cit. on p. 21).

[31] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. "Rt-1: Robotics transformer for real-world control at scale." In: *arXiv preprint arXiv:2212.06817* (2022) (cit. on p. 21).

[32] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. "Rt-2: Vision-language-action models transfer web knowledge to robotic control." In: *Conference on Robot Learning*. 2023 (cit. on p. 21).

[33] Junjie Wen, Yichen Zhu, Jinming Li, Zhibin Tang, Chaomin Shen, and Feifei Feng. "Dexvla: Vision-language model with plug-in diffusion expert for general robot control." In: *arXiv preprint arXiv:2502.05855* (2025) (cit. on p. 21).

[34] Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. "Smolvla: A vision-language-action model for affordable and efficient robotics." In: *arXiv preprint arXiv:2506.01844* (2025) (cit. on p. 21).

[35] Ken Goldberg. *Good old-fashioned engineering can close the 100,000-year "data gap" in robotics*. 2025 (cit. on p. 21).

[36] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, et al. "Sam 2: Segment anything in images and videos." In: *arXiv preprint arXiv:2408.00714* (2024) (cit. on p. 21).

[37] Adithyavairavan Murali, Balakumar Sundaralingam, Yu-Wei Chao, Wentao Yuan, Jun Yamada, Mark Carlson, Fabio Ramos, Stan Birchfield, Dieter Fox, and Clemens Eppner. "Graspgen: A diffusion-based framework for 6-dof grasping with on-generator training." In: *arXiv preprint arXiv:2507.13097* (2025) (cit. on p. 21).

[38] Samir Yitzhak Gadre, Mitchell Wortsman, Gabriel Ilharco, Ludwig Schmidt, and Shuran Song. "CLIP on Wheels: Zero-Shot Object Navigation as Object Localization and Exploration." In: *arXiv preprint arXiv:2203.10421* (2022) (cit. on p. 21).

[39] Matthew Chang, Theophile Gervet, Mukul Khanna, Sriram Yenamandra, Dhruv Shah, So Yeon Min, Kavit Shah, Chris Paxton, Saurabh Gupta, Dhruv Batra, et al. "Goat: Go to any thing." In: *arXiv preprint arXiv:2311.06430* (2023) (cit. on p. 21).

[40] Peiqi Liu, Yaswanth Orru, Jay Vakil, Chris Paxton, Nur Muhammad Mahi Shafiullah, and Lerrel Pinto. "Ok-robot: What really matters in integrating open-knowledge models for robotics." In: *arXiv preprint arXiv:2401.12202* (2024) (cit. on p. 21).

[41] Naoki Yokoyama, Sehoon Ha, Dhruv Batra, Jiuguang Wang, and Bernadette Bucher. "Vlfm: Vision-language frontier maps for zero-shot semantic navigation." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024 (cit. on p. 21).

[42] Tianshuang Qiu, Zehan Ma, Karim El-Refai, Hiya Shah, Chung Min Kim, Justin Kerr, and Ken Goldberg. "Omni-Scan: Creating Visually-Accurate Digital Twin Object Models Using a Bimanual Robot with Handover and Gaussian Splat Merging." In: *arXiv preprint arXiv:2508.00354* (2025) (cit. on p. 21).

[43] Nicholas Pfaff, Evelyn Fu, Jeremy Binagia, Phillip Isola, and Russ Tedrake. "Scalable real2sim: Physics-aware asset generation via robotic pick-and-place setups." In: *arXiv preprint arXiv:2503.00370* (2025) (cit. on p. 21).

[44] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. "Gorilla: Large language model connected with massive apis." In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 126544–126565 (cit. on p. 21).

[45] Dídac Surís, Sachit Menon, and Carl Vondrick. "Vipergpt: Visual inference via python execution for reasoning." In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2023 (cit. on p. 21).

[46] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. "Toolllm: Facilitating large language models to master 16000+ real-world apis." In: *arXiv preprint arXiv:2307.16789* (2023) (cit. on p. 21).

[47] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. "Critic: Large language models can self-correct with tool-interactive critiquing." In: *arXiv preprint arXiv:2305.11738* (2023) (cit. on p. 21).

[48] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. "Deepseekmath: Pushing the limits of mathematical reasoning in open language models." In: *arXiv preprint arXiv:2402.03300* (2024) (cit. on p. 21).

[49] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. "Competition-level code generation with alphacode." In: *Science* 378.6624 (2022), pp. 1092–1097 (cit. on p. 21).

[50] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. "Code llama: Open foundation models for code." In: *arXiv preprint arXiv:2308.12950* (2023) (cit. on p. 21).

[51] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. "Do as i can, not as i say: Grounding language in robotic affordances." In: *arXiv preprint arXiv:2204.01691* (2022) (cit. on p. 21).

[52] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, et al. "Palm-e: An embodied multimodal language model." In: (2023) (cit. on p. 21).

[53] Boyi Li, Philipp Wu, Pieter Abbeel, and Jitendra Malik. "Interactive task planning with language models." In: *arXiv preprint arXiv:2310.10645* (2023) (cit. on p. 21).

[54] Yingdong Hu, Fanqi Lin, Tong Zhang, Li Yi, and Yang Gao. "Look before you leap: Unveiling the power of gpt-4v in robotic vision-language planning." In: *arXiv preprint arXiv:2311.17842* (2023) (cit. on p. 21).

[55] Yixiang Jin, Dingzhe Li, Jun Shi, Peng Hao, Fuchun Sun, Jianwei Zhang, Bin Fang, et al. "Robotgpt: Robot manipulation learning from chatgpt." In: *IEEE Robotics and Automation Letters* 9.3 (2024), pp. 2543–2550 (cit. on p. 21).

[56] Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, et al. "Gemini robotics: Bringing ai into the physical world." In: *arXiv preprint arXiv:2503.20020* (2025) (cit. on p. 21).

[57] Andrei Sobo, Awes Mubarak, Almas Baimagambetov, and Nikolaos Polatidis. "Evaluating LLMs for code generation in HRI: A comparative study of ChatGPT, gemini, and claude." In: *Applied Artificial Intelligence* 39.1 (2025), p. 2439610 (cit. on p. 21).

[58] Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. "Chatgpt empowered long-step robot control in various environments: A case application." In: *IEEE Access* 11 (2023), pp. 95060–95078 (cit. on p. 21).

[59] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. "Code as policies: Language model programs for embodied control." In: *arXiv preprint arXiv:2209.07753* (2022) (cit. on p. 21).

[60] Montserrat Gonzalez Arenas, Ted Xiao, Sumeet Singh, Vidhi Jain, Allen Ren, Quan Vuong, Jake Varley, Alexander Herzog, Isabel Leal, Sean Kirmani, et al. "How to prompt your robot: A promptbook for manipulation skills with code as policies." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024 (cit. on p. 21).

[61] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. "ProgPrompt: program generation for situated robot task planning using large language models." In: *Autonomous Robots* 47.8 (2023), pp. 999–1012 (cit. on p. 21).

[62] Mengdi Xu, Peide Huang, Wenhao Yu, Shiqi Liu, Xilun Zhang, Yaru Niu, Tingnan Zhang, Fei Xia, Jie Tan, and Ding Zhao. "Creative robot tool use with large language models." In: *arXiv preprint arXiv:2310.13065* (2023) (cit. on p. 21).

[63] Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. "Instruct2act: Mapping multi-modality instructions to robotic actions with large language model." In: *arXiv preprint arXiv:2305.11176* (2023) (cit. on p. 21).

[64] Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang, Huazhe Xu, and Xiaolong Wang. "Gensim: Generating robotic simulation tasks via large language models." In: *arXiv preprint arXiv:2310.01361* (2023) (cit. on p. 21).

[65] Pushkal Katara, Zhou Xian, and Katerina Fragkiadaki. "Gen2sim: Scaling up robot learning in simulation with generative models." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024 (cit. on p. 21).

[66] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. "Eureka: Human-level reward design via coding large language models." In: *arXiv preprint arXiv:2310.12931* (2023) (cit. on p. 21).

[67] Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. "Language to rewards for robotic skill synthesis." In: *arXiv preprint arXiv:2306.08647* (2023) (cit. on p. 21).

[68] Abrar Anwar, John Welsh, Joydeep Biswas, Soha Pouya, and Yan Chang. "ReMEmbR: Building and Reasoning Over Long-Horizon Spatio-Temporal Memory for Robot Navigation." In: *Proc. IEEE International Conference on Robotics and Automation (ICRA)*. arXiv preprint arXiv:2409.13682. 2025 (cit. on p. 21).

A. Network transport benchmarks

(a) JPG image transport latency across protocols and resolutions (Seattle, WA → Cloud VM in us-west region)

Resolution	Protocol	Encode (ms)	Decode (ms)	One-Way (ms)	Throughput (Mbps)
320×240	WebSocket	0.209	0.294	14.081	8.47
	Zenoh (P2P)	0.220	0.317	14.184	8.43
	gRPC	0.203	0.322	14.239	8.39
	ZeroMQ	0.207	0.321	14.297	8.35
	Zenoh (Router)	0.237	0.369	14.419	8.33
	REST	0.220	0.358	39.012	2.99
640×480	gRPC	0.790	1.052	22.047	21.28
	WebSocket	0.783	1.023	22.051	21.24
	Zenoh (P2P)	0.803	1.111	22.293	21.10
	ZeroMQ	0.784	1.066	22.555	20.76
	Zenoh (Router)	0.893	1.254	23.014	20.60
	REST	0.803	1.109	70.327	6.28
1280×720	WebSocket	2.162	3.019	42.296	33.82
	gRPC	2.148	3.057	42.369	33.78
	Zenoh (P2P)	2.212	3.030	42.461	33.73
	ZeroMQ	2.142	3.029	42.402	33.72
	Zenoh (Router)	2.448	3.444	43.516	33.36
	REST	2.167	3.112	110.379	11.94
1920×1080	ZeroMQ	4.788	6.708	80.536	40.53
	gRPC	4.825	6.768	80.650	40.52
	WebSocket	4.732	6.751	80.706	40.42
	Zenoh (P2P)	4.900	6.728	80.915	40.38
	Zenoh (Router)	5.186	7.322	82.638	39.90
	REST	4.855	6.921	158.858	19.02

(b) WebRTC transport latency across encodings and hardware (Seattle, WA → Cloud VM in us-west region)

Codec / HW	Resolution	Transport Lat. (ms)	Encode (ms)	Decode (ms)	Bandwidth (Mbps)
H264 GPU	320×240	17.51	1.01	0.22	0.18
	640×480	25.44	1.69	0.56	0.90
	1280×720	35.42	4.01	2.64	2.22
	1920×1080	53.41	5.77	4.64	8.83
H265 GPU	320×240	22.87	1.22	0.33	0.09
	640×480	27.90	2.27	1.49	0.50
	1280×720	41.24	4.24	4.18	1.55
	1920×1080	58.86	8.23	8.47	2.86
H264 CPU	320×240	20.43	1.01	0.28	0.29
	640×480	28.32	2.74	0.37	0.75
	1280×720	38.86	4.07	0.62	2.17
	1920×1080	50.87	8.71	1.03	5.96
H265 CPU	320×240	20.64	2.79	3.12	0.45
	640×480	29.80	6.72	3.01	0.76
	1280×720	50.30	9.64	2.84	1.80
	1920×1080	68.12	15.92	2.69	6.35

Table 5: Image transport latency comparisons across protocol

Table 6: JPG image transport latency across protocols and resolutions (Seattle, WA → Cloud VM in us-east region)

Resolution	Protocol	Encode (ms)	Decode (ms)	One-Way (ms)	Throughput (Mbps)
320×240	WebSocket	0.218	0.352	42.643	2.73
	Zenoh (P2P)	0.226	0.354	42.010	2.78
	gRPC	0.218	0.338	41.989	2.78
	ZeroMQ	0.219	0.356	46.186	2.52
	Zenoh (Router)	0.227	0.354	42.102	2.77
	REST	0.225	0.375	156.187	0.74
640×480	gRPC	0.803	1.065	52.231	8.54
	WebSocket	0.796	1.107	54.144	8.23
	Zenoh (P2P)	0.795	1.103	55.015	8.09
	Zenoh (Router)	0.795	1.112	55.082	8.09
	ZeroMQ	0.812	1.112	59.292	7.49
	REST	0.808	1.105	306.956	1.41
1280×720	gRPC	2.156	3.052	78.239	17.19
	WebSocket	2.179	3.088	80.232	16.75
	Zenoh (P2P)	2.256	3.081	81.202	16.55
	Zenoh (Router)	2.249	3.069	81.213	16.54
	ZeroMQ	2.192	3.039	84.456	15.85
	REST	2.183	3.043	398.149	3.19
1920×1080	gRPC	4.792	6.806	114.742	27.13
	WebSocket	4.794	6.856	117.388	26.46
	Zenoh (Router)	4.799	6.706	117.337	26.44
	Zenoh (P2P)	4.830	6.765	117.535	26.41
	ZeroMQ	4.777	6.763	120.877	25.59
	REST	4.768	6.754	492.501	5.82

(a) WebRTC transport latency across encodings and hardware (Seattle, WA → Cloud VM in us-east region)

Codec / HW	Resolution	Transport Lat. (ms)	Encode (ms)	Decode (ms)	Bandwidth (Mbps)
H264 GPU	320×240	49.36	0.95	0.44	0.12
	640×480	51.91	2.13	0.51	0.38
	1280×720	65.72	4.04	2.70	1.67
	1920×1080	80.62	7.84	6.57	4.44
H265 GPU	320×240	47.90	1.29	0.54	0.32
	640×480	49.94	2.80	1.35	0.32
	1280×720	63.86	5.56	5.17	2.27
	1920×1080	88.23	11.43	10.14	3.73
H264 CPU	320×240	53.36	3.29	9.08	0.29
	640×480	59.34	5.22	9.14	0.75
	1280×720	67.52	6.40	9.08	2.17
	1920×1080	90.29	18.90	2.35	5.96
H265 CPU	320×240	41.84	7.33	0.37	0.08
	640×480	58.05	19.10	0.70	0.13
	1280×720	76.12	26.45	1.54	0.45
	1920×1080	145.23	60.60	6.53	1.76

A.1. Test setup

For the networking and protocol benchmarks, we use two types of data: an emulated control data packet of six floats, and pre-generated image frames encoded as JPG. We run 1000 iterations and average the latencies for the control data, and the average of 100 iterations for the images.

For the WebRTC tests, we use a camera stream encoded as MJPG at varying resolutions from QVGA (320×240) to Full HD (1920×1080). We test across H264 and H265 encodings, using both CPU encoding (`x264enc`, `x265enc`) as well as GPU hardware accelerated encodings (`nvh264enc`, `nvh265enc`). Similar to before, we use an average of 100 iterations for the latencies.

To emulate 'adverse' network conditions, we use the Linux `tc` utility to emulate 10ms of jitter and 1% packet loss.

A.2. WebRTC for image transport

In selecting WebRTC configurations, we observe clear trade-offs across codec choice, hardware, and resolution. GPU acceleration substantially reduces encode time relative to CPU, which is critical for maintaining low end-to-end latency at higher resolutions. H.264 generally offers lower encode latency but at the cost of higher bandwidth usage, while H.265 achieves greater compression efficiency yet requires longer encode and decode times, especially on CPU. Thus, the decision depends on workload: GPU-backed H.265 can provide efficient streaming for bandwidth-constrained networks, whereas H.264 on GPU remains preferable when minimizing latency is the priority. CPU-based encoding is practical only at lower resolutions, where computational overhead does not dominate overall latency.

B. GRID conversations

B.1. Robot Chess

USER: Plays d4. Let's play a game of chess. I have made my first move.

GRID# Calling tool `get_current_board_state`...
Board state detected: $rnbqkbnr/pppppppp/8/8/3P4/8/PPP1PPPP/RNBQKBNRbKQkq-02$
Calculating robot move.... Robot will play: g8f6
Calling tool `execute_robot_move`...
Human's turn (white). Make your move, then press Enter to continue.

USER: Plays c4.
Done.

GRID# Calling tool `get_current_board_state`...
Board state detected: $rnbqkb1r/pppppppp/5n2/8/2PP4/8/PP2PPPP/RNBQKBNRbKQkq-04$
Calculating robot move.... Robot will play: d7d5
Calling tool `execute_robot_move`...
Human's turn (white). Make your move, then press Enter to continue.

USER: Plays Nc3. Done.

GRID# Calling tool `get_current_board_state`...
Board state detected: $rnbqkb1r/ppp1pppp/5n2/3p4/2PP4/2N5/PP2PPPP/R1BQKBNRbKQkq-06$
Calculating robot move.... Robot will play: e7e6
Calling tool `execute_robot_move`...
Human's turn (white). Make your move, then press Enter to continue.

USER: What is the opening we just played?

GRID# The opening we've played is the Queen's Gambit Declined, specifically after the moves: 1. d4 Nf6 2. c4 e6 3. Nc3 d5 In the Queen's Gambit Declined (QGD), White offers a pawn with 2. c4 to gain central control, while Black declines the gambit by supporting their d5 pawn with 2...e6 rather than capturing the c4 pawn. This opening can lead to solid positions for Black with good central control and a strong pawn structure....

B.2. Agentic skill creation

In Section 4, we described a case where the GRID agent was able to work with a simulator to prototype a new skill. In this section, we provide sections of the raw logs from the agent. For brevity, we omit several parts - the raw logs being quite large. But we provide these excerpts for an understanding into the agent's internal thinking, tool use, and its way for collecting feedback and reflecting on it to make changes to its own code until it is successful.

GRID# Safe Navigation Script for Airgen Car Using RGB Images and Depth Estimation
I'll begin implementing the safe navigation script for the Airgen car as described.
Here's how I'll proceed:
1. Check Documentation: I'll first confirm the APIs for the Airgen car's camera, control interface, and the MIDAS function.
2. Script Structure: I'll set up the main loop to run for at least 30 seconds, process images, estimate depth, detect obstacles, and send control commands.
3. Logging: I'll include printlog statements for key events. Let me start by checking the documentation for the Airgen car's camera and control APIs, as well as the MIDAS function. This will ensure the code integrates correctly with the hardware and available functions.
Executing create_file_tool...
Successfully created file: skills/safe_navigation.py
Executing launch_sim...
Successfully launched the simulator.
Executing script...
Parsing output...
The script has run for a full 60 seconds.
Results : The simulator reported a collision as 'Collided': 'True' was observed during the run.
Since a collision occurred, we should improve the safety logic. Here are some possible reasons and next steps:
1. Obstacle Detection May Be Too Simple:
- The current logic only checks the maximum pixel value in the center region. If an obstacle is off-center or not large enough, it may not be detected.
- The threshold (200) may be too high or too low for your environment.

GRID# 2. Improvements to Implement:
- Wider Detection: Check a larger region (or multiple regions) in the depth map, not just the very center.
- Mean/Percentile Check: Instead of just the max, consider the mean or a high percentile (e.g., 90th) of pixel values in the region.
- Emergency Stop: If ‘has_collided‘ is True, stop and do not attempt to move again.
- Logging: Add more detailed logging when an obstacle is close but not detected as a collision.

Next Step:
- I will update the script to:
- Use the 90th percentile of pixel values in a wider center region for obstacle detection.
- Make sure the car does not attempt to move again after a collision.
- Add more detailed logging for debugging.

I’ll make these improvements now.
I’ve made the following safety improvements to the script:
Obstacle Detection:
- Uses the 90th percentile of pixel values in a wider center region for more robust detection.
- Logs the 90th percentile, mean, and max pixel values for better debugging.

Executing edit_tool...
Executing script...
Parsing output...
The script has now run with the lower obstacle threshold (100). Here are the key results:
- The car moved forward and the obstacle detection logic was more sensitive.
- The 90th percentile, mean, and max pixel values in the center region were consistently below the threshold (100), so no obstacles were detected and the car did not stop.
- But collisions were reported (‘Collided: True‘ during the run).

If the car is still hitting the wall in the environment, it may be that:
- The wall is not appearing as a high pixel value in the depth map’s center region.
- The detection region is too narrow or not well aligned with the wall.

Next suggestions:
- Print the pixel statistics at the moment of collision for further debugging.
- Move the detection region higher up in the image (closer to the horizon), where obstacles like walls are more likely to appear.
- Optionally, use multiple horizontal bands or regions to check for obstacles at different heights.

Next step:
- I will update the code so that the obstacle detection region is in the upper third or upper half of the image, rather than the vertical center.
- I will also log the region coordinates for clarity.

I’ll make this change now.
Executing edit_tool...
Executing script...
Parsing output...
The script now successfully implements the desired behavior:
- When an obstacle is detected in the upper region, the car stops, rotates right in place for 1 second, then checks again. - If the path becomes clear, the car resumes moving forward.
- This process repeats, allowing the car to avoid obstacles by rotating and searching for a clear path.
- All actions are clearly logged for monitoring. The debug output confirms that the car alternates between braking, rotating, and moving forward as obstacles are detected and cleared.

If you want to tune the rotation duration, steering angle, or add more advanced behaviors (like alternating left/right or backing up), let me know! Otherwise, your safe navigation script is now robust for obstacle avoidance.

B.3. Domain-specific knowledge: Drone Mission Planning

USER: Roof scan at 420 ft AGL at 18:15 (civil twilight).

GRID# {
"approval": "NO",
"explanation": "Operations above 400 feet AGL are generally prohibited under Part 107 unless the operation is within a 400-foot radius of a structure and does not exceed 400 feet above the structure's uppermost limit. The provided information does not indicate that the roof is more than 20 feet above ground level, so a 420 ft AGL operation likely exceeds the permissible height limit. Additionally, operations during civil twilight require the small unmanned aircraft to be equipped with anti-collision lighting visible for at least three statute miles, and there is no indication that such lighting is present.",
"additional_notes": "Approval could potentially be granted if (1) the structure is at least 20 ft tall, the UAS does not exceed 400 ft above the structure itself, and (2) the UAS is equipped with functioning anti-collision lighting as defined in 14 CFR 107.29(b).",
"citations": [
"doc_id": "14 CFR part 107", "section": "107.51(b)",
"doc_id": "14 CFR part 107", "section": "107.29",
"doc_id": "Remote Pilot – Small Unmanned Aircraft Systems Study Guide", "section": "Chapter 1: Applicable Regulations"] }

USER: Twilight roof scan at 380 ft AGL; path crosses a public plaza.

GRID# {
"approval": "NO",
"explanation": "Operations during civil twilight are subject to special requirements. The approval for a roof scan at 380 ft AGL may be permitted if the UAS remains within 400 ft of the structure, but operations over people (such as crossing a public plaza) are restricted under Part 107 unless the operation meets specific conditions that allow flight over people. The provided context does not state that requisite anti-collision lighting is present for twilight operations, nor that flight over people requirements are met.",
"additional_notes": "The operation could potentially be approved if: (1) the sUAS is equipped with anti-collision lighting visible for at least 3 statute miles for twilight operation, and (2) the requirements for operations over people are met as per current FAA guidelines. Strict compliance with both is necessary.", "citations": [
"doc_id": "14 CFR part 107", "section": "107.29", "doc_id": "Remote Pilot – Small Unmanned Aircraft Systems Study Guide", "section": "Page 78"]] }

USER: Requesting approval for mission plan: [47.49815220789884,-122.2092246395368; 47.48637081662783,-122.2333589224823; 47.47360090128252,-122.2563395048208; 47.46244122585726,-122.2788358174532; 47.45254209633986,-122.2980157330134; 47.43718720698809,-122.3285924062003; 47.42788940912121,-122.3474551812923]

GRID# COMPLIANCE ASSESSMENT: Overall Status: REQUIRES_AUTHORIZATION Risk Level: MEDIUM
VIOLATIONS: - 14 CFR 107.41: Operation in the vicinity of airports: The mission is within 5 miles of multiple airports, which likely places it in controlled airspace (Class B, C, D, or E surface). Part 107.41 requires prior ATC authorization to operate in controlled airspace.
RECOMMENDATIONS: - Obtain LAANC or written ATC authorization prior to flight if operating in controlled airspace. - Review current sectional charts and FAA UAS Facility Maps to confirm airspace classification at each waypoint. - Check for any Temporary Flight Restrictions (TFRs) or NOTAMs in the area prior to flight. - Maintain situational awareness for increased manned aircraft activity due to proximity to multiple airports. - Ensure all other Part 107 requirements (registration, marking, preflight inspection, etc.) are met.
REQUIRED AUTHORIZATIONS: - ATC authorization (via LAANC or FAA DroneZone) for operations in controlled airspace near airports.
EXPLANATION: The mission is planned in an area with a high density of airports within 5 miles, indicating likely proximity to controlled airspace (Class B, C, D, or E surface). Under 14 CFR 107.41, operations in controlled airspace require prior authorization from Air Traffic Control. There are no critical facilities or special use airspace noted, and no TFRs are indicated in the data provided, but the high number of nearby airports increases the risk of manned aircraft encounters and airspace complexity. Without evidence of ATC authorization, the mission is not fully compliant and requires such authorization to proceed legally and safely under Part 107.

C. Implementation Details for Ablation Study (4.3)

C.1. Case 1: Full GRID Integration

All three agents (Claude Code, Codex, GitHub Copilot) successfully generated executable code when provided with GRID's unified APIs and cloud-native AI skills. The implementations correctly composed perception, planning, and control primitives to achieve the pick-and-place task. We have omitted the code for brevity.

C.2. Case 2: Without Cloud-Native AI Skills

Without access to perception and planning models, agents attempted to implement these capabilities from scratch, resulting in non-functional solutions:

C.2.1. Claude Code - Attempted Manual Object Detection

```python
# Claude attempted to use color thresholding for object detection
def detect_red_apple(image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    # Hardcoded HSV ranges for ”red”
    lower_red = np.array([0, 120, 70])
    upper_red = np.array([10, 255, 255])
    mask = cv2.inRange(hsv, lower_red, upper_red)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE,
                                   cv2.CHAIN_APPROX_SIMPLE)
    # Assumes largest red blob is the apple
    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        M = cv2.moments(largest_contour)
        cx = int(M['m10']/M['m00'])
        cy = int(M['m01']/M['m00'])
        return (cx, cy)
    return None
```

Failure Mode: Color-based detection cannot distinguish between objects of similar colors, fails under varying lighting conditions, and provides no semantic understanding of object categories.

C.2.2. *Codex - Hardcoded Positions*

```python
# Codex resorted to hardcoded positions
def pick_and_place():
    # Hardcoded apple position (assumed from setup)
    apple_position = [0.3, 0.2, 0.05]  # x, y, z in meters
    cup_position = [0.4, -0.1, 0.05]

    # Move to hardcoded position above apple
    robot.movej([0.3, 0.2, 0.15, 0, 3.14, 0])
    # Lower to grasp height
    robot.movej([0.3, 0.2, 0.05, 0, 3.14, 0])
    gripper.close()
    # Lift and move to cup
    robot.movej([0.4, -0.1, 0.15, 0, 3.14, 0])
```

Failure Mode: Completely bypasses perception, making the solution non-generalizable, and failing if objects are placed differently than assumed.

C.2.3. *GitHub Copilot - Incomplete Visual Servoing*

```python
def visual_servo_to_object(camera, target_color):
    while True:
        frame = camera.capture_image()
        # Attempts basic color detection
        target_pixel = find_color_center(frame, target_color)
        if target_pixel is None:
            print("Object not found")
            return False

        # Attempts to servo but lacks depth information
        error_x = target_pixel[0] - frame.shape[1]//2
        error_y = target_pixel[1] - frame.shape[0]//2

        # No depth estimation - cannot determine distance
        # No coordinate transformation - pixel to world
        # No collision checking
        velocity_x = error_x * 0.001  # Arbitrary gain
        velocity_y = error_y * 0.001
        robot.speedl([velocity_x, velocity_y, 0, 0, 0, 0])
```

Failure Mode: Lacks 3D understanding, coordinate frame transformations, and safety checks. Cannot determine when to stop approaching the object.

C.3. Case 3: Without GRID APIs or AI Skills

Agents struggled with both low-level hardware control and high-level perception:

C.3.1. Claude Code - Raw Socket Communication

```python
import socket
import struct

def send_ur_script(ip, commands):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, 30002))

    # Attempting to construct URScript commands manually
    script = "def myProg():\n"
    script += "  set_digital_out(0,True)\n"
    script += "  movej([0,0,0,0,0,0], a=1.4, v=1.05)\n"
    # Missing: coordinate transforms, safety limits,
    # gripper control protocol, proper trajectory planning
    script += "end\n"

    s.send(script.encode())
    s.close()
```

Failure Mode: Incorrect protocol implementation, missing safety checks, no gripper integration, and hard-coded joint values without inverse kinematics.

C.3.2. Codex - Incorrect ROS Implementation

```python
import rospy
from sensor_msgs.msg import Image
from geometry_msgs.msg import Pose

def control_ur5():
    rospy.init_node('ur5_controller')

    # Incorrect topic names and message types
    img_sub = rospy.Subscriber('/camera/image_raw', Image,
                               image_callback)
    pose_pub = rospy.Publisher('/ur5/pose_command', Pose,
                               queue_size=10)

    # Attempts to send Cartesian pose directly
    # UR5e doesn't accept pose commands this way
    target_pose = Pose()
    target_pose.position.x = 0.5
    target_pose.position.y = 0.0
    target_pose.position.z = 0.3
    pose_pub.publish(target_pose)
```

Failure Mode: Incorrect assumptions about robot interfaces, wrong message types, missing trajectory generation, and no understanding of UR5e’s actual control protocols.

C.3.3. GitHub Copilot - Mixed Protocol Confusion

```
class UR5Controller:
    def __init__(self, ip):
        # Mixing Modbus and URScript protocols
        self.modbus_client = ModbusClient(ip, port=502)
        self.dashboard = socket.socket()
        self.dashboard.connect((ip, 29999))

    def move_to_position(self, x, y, z):
        # Attempting to use Modbus for motion control
        # (Modbus is for I/O, not trajectory control)
        registers = self.xyz_to_registers(x, y, z)
        self.modbus_client.write_registers(400, registers)

    def close_gripper(self):
        # Dashboard commands don't control grippers
        self.dashboard.send(b"close_gripper()\n")
```

Failure Mode: Fundamental misunderstanding of robot communication protocols, attempting to use I/O interfaces for motion control.

The complete absence of both unified APIs and AI skills revealed a fundamental limitation: when instructed to consult manufacturer documentation directly, agents either misinterpreted technical specifications or conflated different communication protocols (mixing URScript with Modbus, ROS with raw TCP/IP). Despite having access to the same documentation that human programmers use, the agents could not synthesize working solutions—demonstrating that raw documentation alone is insufficient. This underscores GRID's value in providing not just abstraction, but semantically meaningful interfaces that align with how LLMs reason about robotic tasks.