


Università degli Studi di Napoli Federico II
Ph.D. Program in
Information Technology and Electrical Engineering
XXXVIII Cycle

Thesis for the Degree of Doctor of Philosophy

Building Secure Embodied Agents for Personalized Environments

by

Narendra Patwardhan

Advisor: Prof. Carlo Sansone

Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

To my family

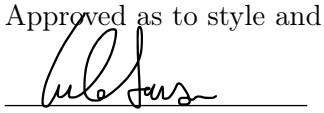
Building Secure Embodied Agents for Personalized Environments

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Narendra Patwardhan

December 2025

Approved as to style and content by



Prof. Carlo Sansone, Advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXVIII cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Finanziato dall'Unione europea
NextGenerationEU

La borsa di dottorato è stata cofinanziata con risorse del
Piano Nazionale di Ripresa e Resilienza, D.M. 352/2022
Missione 4 – Componente 2 – Investimento 3.3 “Dottorati innovativi”
CUP: E66G22000400009

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, December 9, 2025

Narendra Patwardhan

Abstract

Ambient Intelligence envisions environments that anticipate human needs through embedded computational systems. While Large Language Models demonstrate suitable reasoning capabilities, their deployment as embodied agents faces architectural constraints: unsandboxed execution of model-generated code, coupling to specific runtime environments, and personalization requiring either intensive computation or cloud transmission.

This dissertation presents an integrated architecture combining transparent foundation models, formally verified execution substrates, and gradient-free adaptation. The contribution comprises three components: language models at 0.6B, 8B, and 14B parameters trained through staged protocols with modular architectures enabling efficient specialization; an execution framework implementing capability-based security via WebAssembly with formal operational semantics, achieving provable isolation while reducing privileged operations by 95%; and an adaptation mechanism applying local learning rules to low-rank projections, enabling continuous personalization with 12.5× sample efficiency over supervised methods while maintaining 99.6% knowledge retention.

A Smart Couch system validates the architecture, deploying a 0.6B model on embedded hardware for multimodal health monitoring and comfort optimization. The system achieves sub-200ms inference latency and 96.3% task completion accuracy while processing all data locally, demonstrating that formal security, efficient adaptation, and sophisticated reasoning coexist in resource-constrained embodied agents.

Keywords: Embodied Agents, Capability-Based Security, Parameter-Efficient Fine-Tuning, Edge Computing, Ambient Intelligence

Sintesi in lingua italiana

L'Ambient Intelligence immagina ambienti in grado di anticipare le esigenze umane attraverso sistemi computazionali integrati. Sebbene i modelli linguistici di grandi dimensioni dimostrino capacità di ragionamento adeguate, la loro implementazione come agenti incarnati deve affrontare vincoli architettonici: esecuzione non sandboxata del codice generato, accoppiamento con specifici ambienti di runtime e personalizzazione che richiede calcoli intensivi o trasmissione cloud.

Questa tesi presenta un'architettura integrata che combina modelli di base trasparenti, substrati di esecuzione formalmente verificati e adattamento senza gradienti. Il contributo comprende: modelli linguistici con 0,6, 8 e 14 miliardi di parametri addestrati con architetture modulari per specializzazione efficiente; un framework di esecuzione che implementa la sicurezza basata sulle capacità tramite WebAssembly con semantica operativa formale, ottenendo isolamento dimostrabile riducendo le operazioni privilegiate del 95%; e un meccanismo di adattamento che applica regole di apprendimento locale a proiezioni di rango basso, con efficienza 12,5 volte superiore ai metodi supervisionati mantenendo il 99,6% di conservazione delle conoscenze.

Un sistema Smart Couch convalida l'architettura implementando un modello da 0,6 miliardi su hardware integrato per monitoraggio della salute e ottimizzazione del comfort. Il sistema raggiunge latenza di inferenza inferiore a 200ms e accuratezza del 96,3% elaborando tutti i dati localmente, dimostrando che sicurezza formale, adattamento efficiente e ragionamento sofisticato coesistono in agenti incarnati con risorse limitate.

Parole chiave: Agenti incarnati, sicurezza basata sulle capacità, messa a punto efficiente dei parametri, edge computing, intelligenza ambientale

Acknowledgements

The research presented in this dissertation has been supported by the Italian Ministry of University and Research (MUR) under the PNRR program - DM 352, Misura I.3.3 Dottorati innovativi, CUP: E66G22000400006009.

The author's work has been supported by a PhD scholarship funded by SIMAR GROUP s.r.l., Monte Urano (FM), whose collaboration enabled the development and validation of the Smart Couch embodiment system.

The author gratefully acknowledges the research period spent at RealAI EU under the guidance of Tarry Singh, which significantly contributed to the development of the foundations and methodologies presented in this dissertation.

I am profoundly grateful to my advisor, Prof. Carlo Sansone, whose mentorship and vision have shaped not only this research but my development as a scholar. I extend my sincere thanks to Prof. Stefano Marrone for his constant support, insightful guidance, and invaluable feedback throughout this journey.

My colleagues at the PICUS lab have provided an intellectually stimulating environment and unwavering camaraderie. Their discussions, critiques, and encouragement have enriched this work immeasurably.

Above all, I am indebted to my family, whose unconditional love and steadfast support have made me the person I am today. This achievement is as much theirs as it is mine.

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

AOT	Ahead-of-Time
ADC	Analog-to-Digital Conversion
AI	Artificial Intelligence
ALM	Augmented Language Model
AmI	Ambient Intelligence
API	Application Programming Interface
BF16	Brain Float 16-bit
BPE	Byte Pair Encoding
BPMP	Boot and Power Management Processor
CE	Cross-Entropy
CI	Confidence Interval
CKA	Centered Kernel Alignment
CPU	Central Processing Unit

CRA Command Recognition Accuracy

CSV Comma-Separated Values

DVFS Dynamic Voltage and Frequency Scaling

FLOP/s Floating-Point Operations per Second

FSDP Fully Sharded Data Parallelism

FSM Finite State Machine

GB Gigabyte

GDPR General Data Protection Regulation

GiB Gibibyte

GQA Grouped-Query Attention

GPU Graphics Processing Unit

GSM8K Grade School Math 8K

HAL Hardware Abstraction Layer

HBM High-Bandwidth Memory

HIPAA Health Insurance Portability and Accountability Act

HTTP Hypertext Transfer Protocol

I2C Inter-Integrated Circuit

I2S Inter-IC Sound

IFR Intent Fulfillment Rate

INT4 4-bit Integer

INT8 8-bit Integer

IO Input-Output

IoT Internet of Things

IPC Inter-Process Communication

JIT Just-in-Time

JSON JavaScript Object Notation

KiB Kibibyte

KL Kullback-Leibler

KV Key-Value

LLM Large Language Model

LoRA Low-Rank Adaptation

LTS Long-Term Support

LUKS Linux Unified Key Setup

LZW Lempel-Ziv-Welch

MB Megabyte

MFU Model FLOPs Utilization

MMLU Massive Multitask Language Understanding

MRL Matryoshka Representation Learning

mTLS mutual Transport Layer Security

MVP Minimum Viable Product

NLU Natural Language Understanding

NPU Neural Processing Unit

OIG Open Instruction Generalist

OTA Over-the-Air

OTP One-Time Programmable

PEFT Parameter-Efficient Fine-Tuning

PPG Photoplethysmography

PTQ Post-Training Quantization

PTSR Proactive Task Success Rate

PWM Pulse-Width Modulation

QAT Quantization-Aware Training

RAG Retrieval-Augmented Generation

RAM Random-Access Memory

RL Reinforcement Learning

RMSNorm Root Mean Square Normalization

RoPE Rotary Position Embeddings

S1, S2, S3 Stage 1, Stage 2, Stage 3 (Training Stages)

SD Standard Deviation

SDK Software Development Kit

SIMD Single Instruction, Multiple Data

SNR Signal-to-Noise Ratio

SoC System-on-Chip

SPI Serial Peripheral Interface

SQL Structured Query Language

SRAM Static Random-Access Memory

STEM Science, Technology, Engineering, Mathematics

STT Speech-to-Text

SUS System Usability Scale

SwiGLU Swish-Gated Linear Unit

TA Trusted Application

TDP Thermal Design Power

TEE Trusted Execution Environment

TLS Transport Layer Security

TTS Text-to-Speech

UART Universal Asynchronous Receiver-Transmitter

URL Uniform Resource Locator

VOC Volatile Organic Compound

W3C World Wide Web Consortium

WABT WebAssembly Binary Toolkit

WAL Write-Ahead Logging

WASI	WebAssembly System Interface
WAT	WebAssembly Text
WER	Word Error Rate
XML	Extensible Markup Language
XTS	XEX-based tweaked-codebook mode with ciphertext stealing

List of Figures

List of Tables

List of Symbols

The following symbols are used within the thesis

α	A scalar weighting coefficient, typically for loss functions
Δ	An operator representing the change or update to a value
η	The learning rate in an optimization algorithm
$\mathbf{T}$	The agent's toolset of available capabilities
$\mathbf{x}$	An input vector to a model or layer
$\mathscr{L}$	A loss function for model training
μ	The mean or average of a distribution
π	The agent's control policy
ψ	The action decoder or response parser function
ρ	The state encoder or contextualization function
σ	The standard deviation of a distribution
τ	A threshold value for activation or decision-making
A, B	The constituent matrices in a low-rank adaptation

d	The hidden dimension of a neural network model
E	The environment dynamics function (Executor)
H	The interaction history of an agent
h	The number of attention heads in a Transformer model
K	The Key parameter matrix in an attention mechanism
M	The core generative model (e.g., LLM)
r	The rank of a low-rank matrix decomposition
S	The Store in the operational semantics model
V	The Value parameter matrix in an attention mechanism
W	A weight matrix in a neural network

Chapter 1

Introduction

The purpose of computing is insight, not numbers.

Richard Hamming

The vision of Ambient Intelligence (AmI) promises environments that seamlessly anticipate and respond to human needs through embedded computational intelligence.[193, 192, 144, 147, 169, 70] This paradigm envisions proactive systems that operate without explicit commands, demonstrate deep contextual awareness beyond isolated sensor readings,[2] and provide personalized adaptation to individual preferences and patterns.[194, 177] Despite decades of research and the proliferation of Internet of Things (IoT) devices, this vision remains largely unrealized.[153, 89] Contemporary smart home systems offer fragmented capabilities through disconnected devices, each solving narrow problems without the coherent intelligence required for true ambient support.[49, 21]

The emergence of Large Language Models (LLMs) presents an unprecedented opportunity to bridge this gap. These models demonstrate remarkable capabilities in natural language understanding, multi-step reasoning [191], and task planning [83] that could serve as cognitive cores for intelligent environments. However, deploying LLMs as autonomous agents in personal spaces exposes fundamental architectural deficiencies that preclude their use in safety-critical, privacy-sensitive applications. Current agent frameworks suffer from three interconnected limitations that we term

the Security Gap, the Flexibility Gap, and the Personalization Gap.

1.1 Problem Statement

The Security Gap arises from the architectural decision to grant LLM outputs privileged access to system resources. Contemporary frameworks execute LLM-generated code directly through unsandboxed interpreters, creating pathways from probabilistic text generation to critical system operations.[163, 140] This design violates fundamental security principles by treating stochastic model outputs as trusted code.[37] A single prompt injection attack can escalate from text manipulation to arbitrary code execution[203], as demonstrated by vulnerabilities where malicious instructions embedded in processed documents lead to filesystem destruction or data exfiltration.[68, 69]

The Flexibility Gap emerges from tight coupling between agent logic and implementation languages. Current frameworks embed agent capabilities within Python or JavaScript ecosystems, creating monolithic systems that cannot be decomposed or ported across heterogeneous platforms. An agent developed for cloud deployment cannot execute on embedded systems, while edge-optimized implementations cannot leverage cloud resources. This architectural rigidity prevents the distributed deployment essential for Ambient Intelligence, where computation must span from resource-constrained sensors to powerful edge servers.[44, 14]

The Personalization Gap reflects the fundamental mismatch between static pre-trained models and dynamic user contexts.[20] Adaptation through fine-tuning requires computational resources exceeding embedded system capabilities by orders of magnitude[47], while cloud-based personalization necessitates continuous transmission of intimate behavioral data to third parties. Current approaches force an unacceptable trade-off between computational feasibility and privacy preservation, preventing the deep, continuous learning required for genuine personalization.

These gaps are not independent failures but symptoms of a fragmented development paradigm that assembles disparate components without architectural coherence. The prevalent approach combines black-box language models accessed through APIs, general-purpose scripting languages with ambient authority, and ad-hoc sandboxing attempts that provide

probabilistic rather than provable security. This compositional brittleness manifests as systems that are simultaneously over-privileged and under-capable: possessing dangerous access to system resources while lacking the architectural foundations for safe, adaptive operation.

1.2 Motivation

The gaps identified above necessitate a fundamental reconceptualization of how embodied intelligent systems are architected.[18] Rather than incrementally patching existing frameworks, architectures must be developed that integrate security, flexibility, and personalization as first-class design principles from the foundation upward.

The central challenge lies in recognizing that these three gaps are not independent problems amenable to independent solutions. Attempting to retrofit security onto systems designed for cloud deployment, or to add personalization capabilities to models architected without adaptation in mind, merely compounds the architectural debt. Each attempted fix introduces new surfaces for failure: sandboxing mechanisms that can be circumvented, adaptation techniques that leak private data, or security measures that prevent necessary flexibility. The fundamental issue is that contemporary AI agent architectures emerge from composition of components designed in isolation, without consideration for how their interaction properties affect system-level guarantees.

This dissertation is motivated by the conviction that **secure, adaptable, and privately personalized embodied agents for Ambient Intelligence require vertical integration across three architectural layers: transparent foundation models whose behavior can be formally analyzed, a mathematically specified execution substrate providing capability-based security, and neurobiologically inspired adaptation mechanisms enabling continuous on-device learning**.

Vertical integration is defined here as unified design and control over the complete computational stack, from model weights through execution environment to adaptation protocols. This integration is not merely an implementation choice but a fundamental requirement for establishing trust. Security properties cannot be formally verified when core components are

opaque services. Adaptation cannot be efficient when model architectures are inaccessible. Privacy cannot be preserved when personalization requires cloud processing.

The vertically integrated approach will be instantiated through three technical pillars that collectively address the identified gaps:

Foundation Models for Flexibility: A family of transparent language models at 0.6B, 8B, and 14B parameters, developed through a dual-stage training protocol that first establishes broad capabilities then enables efficient task-specific adaptation. These models incorporate novel architectural elements including the Adaptron module for runtime specialization and Matryoshka Representation Learning[97] for hierarchical knowledge distillation.

Formal Substrate for Security: A computational framework built on WebAssembly[73] that replaces ambient authority with capability-based security. The substrate is specified through formal operational semantics, implemented with mathematical rigor, and validated to provide a 95% reduction in system call exposure compared to conventional Python environments while maintaining performance suitable for real-time interaction.

Neuroplastic Adaptation for Personalization: A gradient-free learning mechanism that replaces computationally intensive backpropagation with localized Hebbian update rules.[46, 132] By treating neural activation patterns as learning signals, the method enables continuous adaptation using 12.5× fewer samples than supervised approaches while exhibiting 99.6% retention of pre-existing capabilities.

Research Questions. This dissertation addresses four core research questions that guide the investigation and structure the validation methodology:

RQ1: Can smaller foundation models with achieve competitive performance for agentic tasks while enabling formal verification? Commercial LLMs achieve impressive results but operate as black boxes, preventing formal analysis and requiring cloud connectivity. This question explores whether carefully designed smaller models, when properly integrated within specialized architectures, can provide sufficient reasoning capabilities while maintaining transparency necessary for security verification and edge deployment.

RQ2: How can capability-based security be mathematically guaranteed in embodied agent execution environments? Traditional agent frameworks rely on ambient authority and ad-hoc sandboxing, creating vulnerabilities in safety-critical applications. This question investigates whether formal operational semantics can specify security properties that are both mathematically provable and practically implementable for real-time embodied systems.

RQ3: Can gradient-free neuroplastic adaptation enable efficient on-device personalization while preserving privacy? Existing personalization approaches require computationally intensive gradient-based training or cloud processing, creating barriers for edge deployment and privacy concerns. This question examines whether biologically-inspired local learning rules can achieve continuous adaptation with minimal computational overhead while keeping all processing on-device.

RQ4: Does vertical integration across the computational stack—from foundation models through execution substrates to adaptation mechanisms—yield emergent properties unattainable in fragmented architectures? This overarching question investigates whether holistic system design produces capabilities beyond the sum of individual components, specifically examining latency, security, adaptability, and resource efficiency in physically embodied applications.

These questions are interconnected: answers to RQ1-RQ3 establish individual technical capabilities, while RQ4 evaluates whether their integration produces systems qualitatively superior to compositional alternatives.

1.3 Research Objectives and Methodology

This dissertation pursues four interconnected research objectives that collectively validate the vertical integration thesis:

Objective 1: Develop and validate transparent foundation models optimized for agentic computation, demonstrating that carefully designed smaller models can achieve competitive performance with massive commercial models when properly integrated within specialized architectures.

Objective 2: Design and implement a formally specified execution substrate that provides mathematical guarantees of security properties while supporting the performance requirements of real-time embodied in-

teraction.

Objective 3: Create and evaluate continuous adaptation mechanisms that enable on-device personalization without compromising privacy or requiring computational resources exceeding embedded system capabilities.

Objective 4: Validate the complete architecture through physical embodiment in a safety-critical application, demonstrating that the integrated system achieves the responsiveness, reliability, and adaptability required for Ambient Intelligence.

These objectives will be pursued through a methodology inspired by *design science* paradigm[76]. The overarching methodology integrates three complementary approaches:

Theoretical Development. Formal specifications establish mathematical foundations for security properties and adaptation mechanisms. This includes operational semantics for the execution substrate (Chapter 4) and mathematical analysis of neuroplastic learning dynamics (Chapter 6). Theoretical contributions provide provable guarantees rather than empirical approximations.

System Implementation. Engineering contributions span the complete software stack from model training through runtime deployment. Implementation decisions are guided by theoretical requirements while accommodating practical constraints of embedded deployment. Each component is implemented as production-quality software suitable for real-world evaluation.

Empirical Validation. Both controlled experiments measuring specific properties and real-world deployment assessing holistic system behavior validate the architecture. Quantitative metrics establish performance characteristics while qualitative user studies assess practical utility and usability.

1.4 Contributions

This dissertation makes four primary contributions to the field of embodied artificial intelligence:

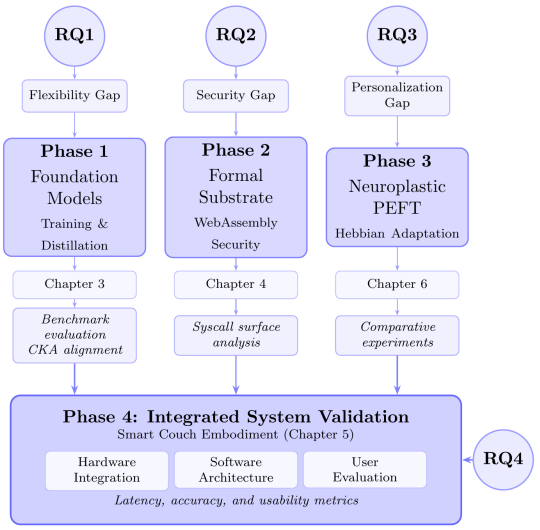


Figure 1.1. Research methodology overview.

Transparent Foundation Models with Adaptive Architecture A family of foundation models specifically architected for agentic computation is presented, spanning three scales to address diverse deployment scenarios. The 14B parameter flagship model establishes semantic understanding through domain-focused pretraining on scientific literature and code repositories. The 8B and 0.6B variants, created through novel distillation techniques incorporating Matryoshka Representation Learning, preserve essential capabilities while enabling edge deployment.

The key innovation is the dual-focus training protocol that separates capability acquisition from adaptation readiness. Initial stages establish broad competence through conventional supervised training. Subsequent stages prepare models for runtime specialization through the Adaptron module, a novel architecture combining Context-Aware Scratchpads with Parameter Attention mechanisms. This separation enables models to serve as flexible cognitive cores (the function $M : \text{String} \to \text{String}$) while remaining amenable to efficient post-deployment adaptation.

Formally Verified Execution Substrate A computational substrate that provides capability-based security through mathematical specification[103] rather than ad-hoc sandboxing is introduced. The substrate is defined via small-step operational semantics, establishing formal properties including memory safety, process isolation, and capability confinement. The implementation maps this formal model to a WebAssembly-based execution environment, leveraging industry-validated sandboxing while maintaining theoretical rigor.

The framework includes a comprehensive toolkit of 20 production-ready tool implementations demonstrating patterns for compound AI systems. These tools, distributed as auditable WebAssembly modules, showcase recursive workflows, parallel exploration strategies, and multi-model orchestration while maintaining security guarantees. Performance evaluation demonstrates that security benefits justify modest overhead, with the framework achieving interactive latency despite sandboxing constraints.

Gradient-Free Neuroplastic Adaptation Neuroplastic PEFT is presented as a novel adaptation mechanism that enables continuous learning without gradient computation. The method applies Hebbian learning principles directly to low-rank adaptation matrices, using neural activation patterns from live interactions as training signals. This approach eliminates the computational bottleneck of backpropagation, enabling real-time adaptation on resource-constrained devices.

The theoretical foundation establishes equivalence between Parameter Attention in the Adaptron architecture and low-rank projections in conventional PEFT, enabling neuroplastic updates to be applied uniformly across adaptation mechanisms. Empirical validation demonstrates dramatic efficiency gains: convergence after 5-7 interactions versus 75 for supervised approaches, with negligible catastrophic forgetting. The method preserves privacy by operating entirely on transient activations without storing or transmitting user data.

Validated Embodiment in Smart Couch Application The complete architecture is validated through deployment in a Smart Couch system that integrates health monitoring, comfort optimization, and emergency detection capabilities. The embodiment demonstrates that vertical

integration enables emergent properties impossible in fragmented architectures: sub-200ms response latency running entirely on-device, 96.3% intent fulfillment accuracy with a 600MB model, and successful continuous adaptation to user-specific preferences.

The implementation required careful co-design of sensing infrastructure, actuation systems, and cognitive architecture. Hardware design decisions, from sensor selection to actuator configuration, were guided by the capabilities and constraints of the software stack. This holistic approach validates that trustworthy embodied intelligence demands not merely better algorithms but fundamental rethinking of system architecture from silicon to semantics.

1.5 Roadmap

The dissertation is organized to systematically build the case for vertical integration, progressing from component development through system integration to deployment validation.

Chapter 2 will survey the technical foundations for secure, efficient, and adaptive embodied agents, examining existing approaches to edge deployment of foundation models, capability-based security architectures, and privacy-preserving personalization. Key achievements and fundamental limitations in each area will be identified that motivate the vertically integrated approach developed in subsequent chapters.

Chapter 3 will present the development of transparent foundation models through a dual-focus methodology. The first part will detail the training pipeline, model architectures, and distillation techniques that create the model family. The second part will introduce adaptation mechanisms, including the Adaptron module and retrieval-augmented generation[104], that transform static models into flexible cognitive cores.

Chapter 4 will introduce the formally specified execution framework, progressing from abstract computational model through WebAssembly implementation to practical tool development. Security properties will be established through mathematical proof, performance characteristics will be demonstrated through systematic evaluation, and implementation patterns for compound AI systems will be provided.

Chapter 5 will detail the Smart Couch implementation, from hardware

architecture through software integration to user evaluation. Design decisions, engineering challenges, and validation results will be documented that demonstrate the feasibility of deploying sophisticated AI on embedded systems while maintaining safety and privacy guarantees.

Chapter 6 will present the neuroplastic adaptation mechanism, establishing theoretical foundations, implementation details, and empirical validation. The chapter will demonstrate how gradient-free learning enables continuous personalization while preserving privacy and computational efficiency.

Chapter 7 will synthesize contributions, acknowledge limitations, and outline future research directions. Broader implications for trustworthy AI development and the path toward realizing Ambient Intelligence will be reflected upon.

The chapters that follow present a comprehensive investigation into architectures for trustworthy embodied intelligence. Each component - transparent models, formal security, and neuroplastic adaptation - addresses specific technical challenges while contributing to a coherent vision of AI systems that operate as personal capabilities rather than remote services. Through rigorous development, systematic validation, and real-world deployment, it will be demonstrated that the path to Ambient Intelligence requires not incremental refinement of existing approaches, but fundamental architectural innovation grounded in principles of transparency, formal verification, and privacy-preserving adaptation.

Chapter 2

Related Work

If I have seen further it is by
standing on the shoulders of giants.

Isaac Newton

The quest for intelligent agents capable of operating securely and adaptively in personal environments draws on diverse research traditions spanning foundation models, secure computation, embodied cognition, and privacy-preserving machine learning.[23, 57] This chapter surveys the intellectual landscape from which this work emerges, examining both the achievements that enable current progress and the limitations that motivate the architectural innovations presented here. The survey is organized into three primary areas: foundation models and their deployment constraints in resource-limited settings, security architectures for embodied systems with emphasis on capability-based approaches, and privacy-preserving personalization mechanisms.

Key Definitions and Scope. This dissertation operates at the intersection of several technical domains. To establish precise terminology and delineate scope, the following definitions specify core concepts that structure subsequent technical development:

Ambient Intelligence (AmI): A paradigm for computational environments characterized by three essential properties: (1) *embedded sensing* where computational intelligence resides within the physical environment rather than requiring users to explicitly engage computing devices, (2) *con-*

text awareness enabling systems to perceive and interpret environmental state including user presence, activity patterns, and situational factors, and (3) *proactive assistance* where systems initiate helpful actions based on inferred needs rather than waiting for explicit commands. AmI extends beyond simple automation by requiring semantic understanding of human activities and adaptive responses to changing contexts.

Embodied Intelligence: Computational agents that maintain bidirectional coupling with physical environments through sensors (perception) and actuators (action), where the agent's cognitive processes are fundamentally shaped by and dependent upon this physical grounding. Distinguished from purely software agents, embodied intelligence systems must address challenges including: sensor noise and ambiguity, actuator dynamics and physical constraints, real-time response requirements, and safety considerations inherent in physical world interaction. This work specifically examines embodied agents deployed in domestic environments where close proximity to humans elevates safety and trustworthiness requirements.

Personalized Environments: Physical spaces where computational systems adapt behavior, interface modalities, and service provision to individual user preferences, learned patterns, and contextual needs. Personalization in embodied contexts differs from web personalization in critical dimensions: (1) *temporal continuity* where adaptation must occur across extended interaction sessions spanning days or weeks, (2) *privacy sensitivity* where personal behavioral data remains physically proximate to users rather than aggregated in remote datacenters, and (3) *multi-modal integration* where personalization spans diverse interaction channels including speech, gesture, environmental controls, and ambient displays.

Edge Deployment: Computational workload execution on resource-constrained devices physically proximate to data sources and users, rather than on remote cloud infrastructure. For AI systems, edge deployment implies: (1) *compute constraints* where models must execute within power, thermal, and memory envelopes of embedded processors, (2) *latency requirements* demanding sub-100ms response for real-time interaction, (3) *privacy preservation* where sensitive data never leaves the device, and (4) *operational autonomy* where systems function during network unavailability.

Scope Delimitation: This dissertation focuses specifically on embodied agents for domestic AmI applications where single users or small household groups interact with stationary, powered furniture embedding computational intelligence. The work explicitly excludes: mobile robots requiring navigation and manipulation planning, multi-occupant public spaces with complex social dynamics, and safety-critical medical or industrial applications requiring regulatory certification. The target deployment platform comprises embedded systems in the 5-50W power envelope, typified by NVIDIA Jetson or similar edge AI accelerators, distinguishing this work from both microcontroller-scale IoT (insufficient compute for LLMs) and datacenter-scale infrastructure (where constraints motivating the architecture do not apply).

2.1 Foundation Models at the Edge

The emergence of large language models has fundamentally transformed the landscape of artificial intelligence, yet their deployment in embodied agents faces severe practical constraints. Although GPT-4[22, 3], Claude[139], and similar models demonstrate remarkable reasoning capabilities, their computational demands, often requiring hundreds of gigabytes of memory and specialized hardware accelerators, make them unsuitable for edge deployment in personal devices. The architectural choices that enable their impressive performance, including attention mechanisms scaling quadratically with sequence length and parameter counts in the hundreds of billions, create an inherent tension between capability and deployability that current research only partially addresses.

Recent efforts toward model compression have produced notable successes in reducing computational footprints while preserving task performance.[7] Knowledge distillation techniques, pioneered for language models through teacher-student architectures, demonstrate that smaller models can approximate the behavior of larger ones when trained on their outputs [77, 152]. However, distillation inherently limits the student model to a subset of the teacher's capabilities, and the process itself requires access to the large model during training, perpetuating dependence on centralized computational resources. Quantization approaches, reducing numerical precision from 32-bit floating point to 8-bit or even 4-bit representations,

achieve dramatic reductions in memory usage [42], but introduce subtle degradations in reasoning quality that compound during multistep inference chains characteristic of agentic behavior.

The MobileLLM architecture [110] represents a significant advance in designing models specifically for edge deployment rather than adapting desktop-scale architectures. By employing grouped-query attention, shared embeddings, and immediate block-wise weight sharing, these models achieve sub-billion parameter counts while maintaining coherent text generation. However, even these optimized architectures [84] struggle with the complex reasoning tasks required for embodied agents, particularly when processing multimodal inputs or maintaining long-term context across extended interactions. The fundamental tension persists: models small enough for edge deployment lack the emergent capabilities that make large models valuable for agentic applications.

Structured state-space models[72, 71, 61] offer an alternative computational paradigm that challenges the dominance of transformer architectures. By reformulating sequence modeling through linear state space equations, models like Mamba achieve linear rather than quadratic scaling with sequence length while maintaining competitive performance on language tasks. These efficiency gains become particularly pronounced for the long contexts typical of embodied agents maintaining interaction histories, environmental observations, and user preferences. However, the relative immaturity of the state-space model ecosystem, which lacks the extensive pretraining datasets and fine-tuning recipes developed for transformers, limits their immediate applicability despite promising theoretical properties.

The challenge of deploying foundation models extends beyond raw computational constraints to encompass the broader system architecture required for robust operation. Current approaches typically rely on cloud-based inference, transmitting user queries to remote servers for processing. This architecture introduces multiple failure modes: network latency disrupts real-time interaction, bandwidth limitations constrain multimodal input processing, and service availability depends on external infrastructure beyond user control. Moreover, the privacy implications of transmitting personal conversations, environmental observations, and behavioral patterns to corporate servers create fundamental tensions with the inti-

mate nature of embodied agents in personal spaces.[170, 204]

Edge-cloud hybrid architectures attempt to balance local responsiveness with cloud-scale capabilities by partitioning computation between devices and servers. Small local models handle routine tasks and initial processing, escalating to cloud resources for complex reasoning or knowledge-intensive queries. Although this approach reduces latency for common interactions, it perpetuates the fundamental dependency on external infrastructure and introduces complex coordination challenges. Determining which queries require cloud processing, managing state synchronization across distributed components, and handling graceful degradation when cloud services become unavailable remain open research problems with significant implications for system reliability.

The adaptation of foundation models to specific domains and user preferences presents additional challenges in resource-constrained settings. Fine-tuning large models requires substantial computational resources and risks catastrophic forgetting of pre-trained knowledge. Parameter-efficient fine-tuning methods, including LoRA (Low-Rank Adaptation)[81] and prefix tuning[106], reduce the number of trainable parameters by introducing small adapter modules or learnable prompts.[80, 101] These techniques achieve impressive results on benchmark tasks but struggle with the continuous, multi-objective adaptation required for personalized agents that must simultaneously maintain general knowledge, adapt to user preferences, learn new skills, and preserve privacy constraints.

Recent developments in retrieval-augmented generation offer a pathway to enhance small models with external knowledge without increasing parameter counts. By conditioning generation on retrieved documents[104, 15, 85], these systems access vast knowledge bases while maintaining compact model sizes. However, the retrieval process itself introduces computational overhead, and the quality of generation depends critically on the retrieval accuracy. For embodied agents, the challenge intensifies as relevant knowledge spans personal interaction histories, environmental observations, learned preferences, and general world knowledge, requiring sophisticated indexing and retrieval mechanisms that themselves consume computational resources.

The emergence of mixture-of-experts architectures provides another approach to scaling model capability without proportionally increasing infer-

ence cost[162, 58]. By routing inputs to specialized subnetworks, these models achieve sparse activation where only a subset of parameters processes each input. This architectural pattern aligns well with embodied agents that encounter diverse task types, from natural language interaction to environmental control to preference learning. However, the routing mechanism itself becomes a critical design choice, and poor routing decisions can dramatically degrade performance. Moreover, the total model size remains large even if inference is sparse, creating storage challenges for edge deployment.

2.2 Security for Embodied Agents

The integration of intelligent agents into personal environments demands rigorous security architectures that protect both user privacy and system integrity. Traditional security models, developed for desktop computing and client-server architectures, prove inadequate for embodied systems that continuously perceive their environment, adapt to user behavior, and execute actions in physical space. The attack surface expands dramatically when agents can observe private conversations, access personal documents, control home appliances, and learn intimate behavioral patterns. This expanded threat model requires fundamental reconsideration of security architectures, moving beyond perimeter defense and access control lists toward capability-based security and formal verification. [41]

Capability-based security[103], rooted in the principle of least privilege, provides a theoretical foundation for secure agent architectures. Unlike traditional access control lists that associate permissions with identities, capabilities bundle access rights with unforgeable references to resources. This approach naturally supports the dynamic, fine-grained permission management required for agents that load tools, access sensors, and interact with external services. The Cambridge CAP computer and subsequent capability systems demonstrated the feasibility of hardware-enforced capabilities, but modern implementations typically rely on software-based approaches using cryptographic techniques or language-based enforcement.[120]

The WebAssembly specification represents a practical realization of capability-based security principles in a portable, performant execution environment. Originally designed for web browsers, WebAssembly pro-

vides memory safety through linear memory isolation, control flow integrity through structured control flow, and resource limits through deterministic execution. [73, 189] These properties make it an attractive substrate for executing untrusted code, such as dynamically loaded tools or third-party extensions.[168] The WebAssembly System Interface (WASI) extends these guarantees to system calls, mediating access to files, networks, and other resources through capability-based APIs. For embodied agents, WebAssembly offers a principled approach to tool isolation that prevents malicious or buggy tools from compromising system integrity.

The application of formal methods to agent architectures remains limited despite their critical role in safety-critical deployments. Temporal logic specifications can express complex security properties, such as ensuring that private information never flows to untrusted tools or that safety invariants are maintained during environmental control. Model checking [31, 91, 164] techniques can verify these properties for finite-state systems, but the complexity of modern agents, incorporating neural networks and continuous adaptation, challenges traditional verification approaches.[137] Recent work on neural network verification focuses primarily on robustness properties, such as resistance to adversarial examples, rather than the behavioral properties relevant to agent security.

Information flow control provides a complementary approach to capability-based security by tracking how information propagates through systems.[40, 125] Taint analysis marks data from untrusted sources and prevents it from influencing security-critical decisions. For embodied agents, this approach could prevent sensor data from untrusted devices from affecting actuator commands or ensure that personal information does not leak to cloud services. However, the dynamic nature of agent behavior, where information flows depend on learned models and user interactions, complicates static analysis. Dynamic information flow tracking incurs runtime overhead that may be prohibitive for resource-constrained edge devices.

The security challenges of machine learning models themselves introduce additional complexity beyond traditional software security.[175] Adversarial examples[65] demonstrate that imperceptible perturbations to inputs can cause misclassification, with potentially severe consequences for embodied agents making safety-critical decisions. Backdoor attacks during training can introduce hidden functionality triggered by specific inputs.

Model extraction attacks can steal intellectual property or enable adversaries to craft more effective attacks. These ML-specific vulnerabilities require new defensive techniques, from adversarial training to differential privacy, that must be integrated into the broader security architecture.

Secure multi-party computation[64] and homomorphic encryption[62] offer theoretical solutions for privacy-preserving computation on sensitive data. These techniques would enable agents to process encrypted sensor data or user preferences without accessing plaintext, providing strong privacy guarantees. However, the computational overhead—often several orders of magnitude—makes them impractical for real-time agent operation. Selective application to particularly sensitive operations, such as processing medical data or financial information, may be feasible, but determining which operations require such protection without leaking information through the protection pattern itself remains challenging.

The challenge of secure software updates[12] for deployed agents intersects with both security and adaptability requirements. Traditional update mechanisms that replace entire software images disrupt continuous operation and risk losing learned adaptations. Incremental updates that modify running systems introduce complex versioning and compatibility challenges. The ability to rollback problematic updates without losing user data or learned preferences requires sophisticated state management. Moreover, the update mechanism itself becomes a critical attack vector, as compromising the update pipeline would allow adversaries to install malicious code on deployed devices[151].

Trusted execution environments, implemented through hardware features like Intel SGX[34] or ARM TrustZone[105], provide isolated execution contexts with hardware-enforced protection. These environments could protect sensitive agent components, such as preference learning algorithms or cryptographic operations, from compromise even if the main system is breached. However, trusted execution environments typically offer limited computational resources and restricted functionality, constraining their applicability to full agent systems. Side-channel attacks through timing, power, or electromagnetic emanations remain viable even with hardware protection, requiring additional countermeasures that further impact performance.

2.3 Privacy-Preserving Adaptation

The promise of embodied agents lies partly in their ability to adapt to individual users, learning preferences, routines, and communication styles through continuous interaction. This personalization requires processing intimate behavioral data, from daily schedules to emotional responses, raising profound privacy concerns. The tension between effective personalization and privacy preservation drives research into techniques that enable learning from user data without compromising confidentiality.[52] These approaches must address not only the technical challenge of privacy-preserving computation but also the broader system design questions of data governance, user control, and regulatory compliance.

Differential privacy[54, 53] provides a mathematically rigorous framework for quantifying and limiting privacy loss during data analysis. By adding calibrated noise to computations, differential privacy ensures that individual records have limited influence on outcomes, providing plausible deniability. For personalized agents, differential privacy could protect user data during preference learning or when contributing to federated learning systems[1, 116]. However, the noise required for strong privacy guarantees often degrades model utility, particularly for the small datasets typical of individual users. The composition of multiple differentially private operations, as would occur during continuous learning, leads to privacy budget exhaustion that eventually prevents further adaptation.

Federated learning[94] enables collaborative model training without centralizing data, allowing agents to benefit from collective experience while maintaining data locality. Each agent trains on local data and shares only model updates with a coordination server, which aggregates updates to improve a global model. This approach preserves privacy in the sense that raw data never leaves user devices, but model updates themselves can leak information about training data. Secure aggregation protocols using cryptographic techniques[13] can prevent the server from observing individual updates, but these protocols introduce communication overhead and complexity that challenge deployment at scale.

The phenomenon of catastrophic forgetting[60] poses a fundamental challenge to continuous learning in neural networks. When trained on new tasks, networks tend to rapidly lose performance on previously learned

tasks unless specific mechanisms preserve prior knowledge.[115] For personalized agents, this manifests as forgetting user preferences when adapting to new situations or losing general capabilities when specializing to individual users. Elastic weight consolidation and similar approaches slow forgetting by penalizing changes to important weights, but determining importance remains heuristic[93]. Memory replay techniques that periodically retrain on stored examples preserve performance but require maintaining and securing historical data, creating privacy risks.

Gradient-free optimization methods offer an alternative approach to personalization that avoids some privacy risks associated with gradient-based learning. Evolutionary strategies[149], genetic algorithms, and other black-box optimization techniques adapt models using only fitness evaluations rather than detailed gradient information. This approach naturally preserves privacy since the optimization process does not require access to raw data or intermediate activations. However, gradient-free methods typically require many more iterations to converge and may struggle with high-dimensional parameter spaces characteristic of neural networks[172]. The trade-off between privacy preservation and adaptation efficiency remains an active area of investigation.

Split learning partitions neural networks between edge devices and servers[185, 184], with each party computing only part of the forward and backward passes. This approach reduces computational load on edge devices while limiting the information shared with servers to intermediate activations rather than raw data. For embodied agents, split learning could enable leveraging cloud-scale models while maintaining privacy. However, intermediate activations still contain significant information about inputs, and adversaries with knowledge of the model architecture can potentially reconstruct sensitive data. The communication overhead of transmitting activations for each inference also impacts real-time performance.

The concept of machine unlearning[16] addresses the requirement, increasingly mandated by privacy regulations, that systems must be able to forget specific data upon request. For traditional databases, deletion is straightforward, but machine learning models trained on data maintain implicit representations that persist after training data deletion. Exact unlearning requires retraining from scratch without the deleted data, which is computationally prohibitive for continuously learning agents. Approxi-

mate unlearning techniques attempt to remove data influence through gradient updates or model editing, but providing guarantees about the completeness of forgetting remains challenging. The interaction between unlearning and personalization creates particular tensions, as removing data points may significantly impact model behavior for individual users.[63]

Privacy-preserving record linkage[155] and entity resolution enable agents to connect information across databases without revealing identities[183]. These techniques would allow agents to access relevant external knowledge, such as medical databases or financial records, while maintaining confidentiality. Cryptographic protocols using secure multi-party computation or private set intersection can determine matches without exposing non-matching records. However, the computational complexity and communication overhead of these protocols limit their applicability to real-time agent operations. Moreover, the mere fact that queries are being made can leak information about user interests or conditions.

The emergence of self-supervised learning reduces dependence on labeled data, enabling agents to learn from raw environmental observations and user interactions. Contrastive learning approaches that maximize agreement between augmented views of the same data[28] have achieved remarkable success in vision[167] and language domains[56]. For embodied agents, self-supervised learning could enable continuous improvement without explicit user feedback, learning representations from the natural correlation between sensory modalities and temporal sequences. However, the computational requirements of contrastive learning, particularly the large batch sizes needed for effective negative sampling, challenge deployment on edge devices.

The integration of causal reasoning[136] into personalization systems promises more robust and interpretable adaptation. By learning causal models of user behavior rather than mere correlations, agents can better predict the effects of interventions and adapt to distribution shifts[156]. Causal discovery from observational data remains challenging, particularly in the presence of hidden confounders and feedback loops characteristic of human-agent interaction. The sample efficiency of causal learning is typically poor, requiring many observations to distinguish causal relationships from spurious correlations. For personalized agents with limited interaction data per user, this presents a fundamental limitation that may require

leveraging population-level causal knowledge.

Meta-learning approaches that learn to learn promise rapid adaptation to new users with minimal data. By training on a distribution of tasks, meta-learning algorithms acquire inductive biases that enable few-shot learning in new domains. Model-Agnostic Meta-Learning (MAML) and its variants[59, 129, 165] optimize for parameters that can be quickly fine-tuned to new tasks with a few gradient steps. For personalized agents, meta-learning could enable rapid adaptation to new users by leveraging experience across the user population. However, the computational cost of meta-learning, requiring second-order gradient computations and multiple adaptation steps, challenges deployment on resource-constrained devices. Moreover, the assumption that new users come from the same distribution as training users may not hold for diverse populations.

The challenge of evaluating personalized systems extends beyond traditional metrics to encompass privacy preservation, adaptation quality, and user satisfaction.[6] Standard benchmarks typically measure task performance on fixed datasets, failing to capture the dynamics of continuous learning or the subtleties of personalization. Simulation environments that model user behavior and preferences enable systematic evaluation but may not reflect real-world complexity. User studies provide ecological validity but raise ethical concerns about experimenting with privacy-sensitive systems.[107] The development of evaluation frameworks that balance rigor, realism, and ethics remains an open challenge with significant implications for research progress.

The regulatory landscape surrounding data protection and algorithmic accountability increasingly constrains system design. The European General Data Protection Regulation (GDPR)[186] grants users rights to explanation[187], rectification, and erasure that challenge traditional ML architectures. The California Consumer Privacy Act (CCPA) and similar legislation extend these requirements globally. For embodied agents processing intimate personal data, compliance requires not just technical mechanisms but also interpretable audit trails, user controls, and transparent data governance. The tension between regulatory requirements and technical capabilities drives research into interpretable models, explainable AI, and privacy-preserving systems that may sacrifice performance for compliance.

The integration of these diverse research threads—efficient foundation models, secure execution environments, and privacy-preserving personalization—remains largely unexplored. Current systems typically address each concern in isolation, bolting security onto existing architectures or adding privacy as an afterthought. The fundamental insight motivating this work is that trustworthy embodied intelligence requires holistic design where security, efficiency, and privacy are foundational rather than supplementary. The architectural principles that will be developed in subsequent chapters emerge from recognizing that the challenges surveyed here are not independent problems to be solved separately but interconnected aspects of a unified system design challenge. By reconsidering the entire stack from hardware abstraction through application logic as an integrated whole, it will be demonstrated that secure, adaptive, and private personal agents are not merely aspirational but achievable through principled engineering.

Chapter 3

Foundation Models for Embodied Intelligence

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W. Dijkstra

The opaque nature of large language models accessed via commercial APIs erects substantial barriers to both rigorous scientific investigation and the engineering of high-assurance systems for embodied agents. From a research standpoint, this paradigm fundamentally precludes mechanistic interpretability; it becomes impossible to probe a model's internal activations, analyze its attention patterns, or directly correlate specific subsets of training data with emergent behaviors. The absence of versioning guarantees and transparency regarding unannounced updates to architecture or training data undermines experimental reproducibility, a cornerstone of the scientific method. For systems development in security-sensitive applications such as Ambient Intelligence, this opacity prevents formal verification of model behavior and introduces intractable risks related to data privacy and adversarial vulnerabilities, as the model's internal defenses cannot be audited, customized, or hardened for specific threat models.

Innovation in agentic AI is frequently contingent upon the ability to deeply customize models for specific tasks and operational domains, a ca-

pability severely impeded by the lack of architectural control in commercial APIs. By offering generalized, one-size-fits-all solutions, these services stifle progress in agentic safety and optimization. The inability to modify a model's architecture prevents research into more efficient computational structures or the integration of novel mechanisms tailored to specific data modalities. More critically, the lack of access to pre-training data and its associated methodology renders fine-tuning a superficial process, as it becomes impossible to perform continued pre-training on specialized corpora to adjust the model's core knowledge base for specific domains.

Beyond constraints on scientific inquiry, reliance on third-party APIs introduces significant economic and data privacy implications that are untenable for agents deployed in personalized, data-sensitive environments. The prevalent per-token pricing model, while suitable for low-volume prototyping, becomes economically infeasible for high-throughput services or research requiring millions of model queries. This dependency creates operational fragility, as users have no control over service parameters such as rate limits or sudden price modifications. From a privacy perspective, the requirement to transmit continuous streams of personal data to third parties constitutes an unacceptable risk. For sectors governed by regulations such as HIPAA or GDPR, or for applications handling sensitive information, external API usage is often prohibited outright. On-premises or private cloud deployment of self-owned models is therefore a mandatory prerequisite for ensuring data sovereignty and maintaining a robust security posture.

The principle of *model sovereignty* is proposed as a necessary condition for developing AI systems that are trustworthy, robust, and sustainable over the long term. This concept encompasses complete control over the entire model lifecycle, including data provenance, architectural design, training infrastructure, and inference deployment. Sovereignty enables true scientific auditability, allowing deep inspection required to understand and mitigate model biases, identify failure modes, and remediate security vulnerabilities. It ensures operational independence, insulating systems from the unpredictable technical roadmaps and business decisions of external providers. Most importantly, it aligns responsibility with capability, ensuring that entities deploying AI systems retain full accountability, which is only possible with complete control over their construction and opera-

tion.

In direct response to these limitations, this chapter introduces a new family of foundational language models conceived from first principles to embody the ideal of model sovereignty. The core design philosophy creates a *tiered family of models* rather than a single monolithic entity, with each member engineered to serve distinct computational and application niches within an Ambient Intelligence framework. This spans from large-scale cloud infrastructure requiring powerful reasoning capabilities to resource-constrained edge devices demanding maximal efficiency. The approach emphasizes transparency in data curation, clarity in architecture, and efficiency in implementation. By constructing these models systematically, this research produces fully auditable and controllable software artifacts that provide stable and secure foundations upon which complex, verifiable, and domain-specific agentic systems can be constructed and scientifically evaluated.

While architectural recipes for creating smaller, more efficient models are increasingly accessible,[110] the primary barrier to their effective use remains the immense computational cost of pre-training. Model performance is critically dependent on training with vast datasets, often exceeding one trillion tokens, a scale that renders pre-training from scratch computationally and economically infeasible for most academic and independent research efforts. To quantify this challenge, the approximate time is modeled required for training using the relationship between model size, dataset size, and available computing resources:

$$T_{\text{train}} = \frac{6 N_p N_t}{N_{\text{gpu}} T_{\text{gpu}} \eta_{\text{mfu}}} \tag{3.1}$$

where N_p denotes the number of model parameters, N_t the number of training tokens, N_{gpu} the total GPU count, T_{gpu} the peak throughput per GPU in FLOP/s, and η_{mfu} the Model FLOPs Utilization (typically 0.3-0.6). The factor of 6 approximates FLOPs required per parameter per token (3 for forward pass, 3 for backward pass). Consider pre-training a 15 billion parameter model on one trillion tokens. The total computational requirement is approximately 9×10^{22} FLOPs. On a high-end academic server with four NVIDIA A100 GPUs (each providing 3.12×10^{14} FLOPs/s for BF16 operations) at 50% MFU, the estimated training time exceeds

1,669 days. Even with eight NVIDIA H100 GPUs (9.89×10^{14} FLOPs/s each), this reduces to 263 days of continuous operation. While technologically achievable, such extensive training coupled with hardware acquisition and operational costs are prohibitive for academic research. This analysis demonstrates that the strategic starting point must be the careful selection and adaptation of an existing open-source base model possessing desired architectural traits.

The foundation of all modern Large Language Models is the Transformer architecture, introduced by Vaswani et al.[182] This architecture represented a paradigm shift in sequence processing, moving away from recurrent structures that processed tokens sequentially. The core innovation is the self-attention mechanism, which allows models to weigh the importance of all tokens in the input sequence simultaneously, enabling capture of complex, long-range dependencies. Modern generative LLMs are predominantly based on decoder-only variants of this architecture. Unlike encoder-decoder or encoder-only models[43], the decoder-only structure is highly efficient for next-token prediction, the fundamental operation of text generation. Its causal self-attention mechanism ensures that predictions for a given token depend only on preceding tokens, making it naturally suited for sequential data generation.[142, 143, 22]

To construct a high-performance agentic foundation, two modern architectural features were identified as non-negotiable: Grouped-Query Attention (GQA)[5] and Rotary Position Embeddings (RoPE).[171] GQA is a critical optimization for inference efficiency that directly addresses the memory bottleneck of the Key-Value cache. In standard multi-head attention, each attention head maintains its own key and value projections. GQA modifies this by allowing multiple query heads to share a single key-value head, drastically reducing KV cache size, the dominant consumer of VRAM during inference, especially with long contexts. This enables processing of significantly longer sequences and accelerates inference speed by lowering memory bandwidth requirements.

The adoption of RoPE is essential for enabling robust long-context reasoning and avoiding brittleness associated with absolute positional embeddings. Absolute positional embeddings assign fixed embeddings to each position in a sequence, failing to generalize effectively beyond maximum sequence lengths seen during training, often leading to catastrophic perfor-

mance decline. In contrast, RoPE encodes positional information relatively by applying rotation matrices to query and key vectors, preserving relative distance between tokens. This intrinsic property allows models to extrapolate understanding of sequence order to lengths far exceeding training data. For agents operating in dynamic environments, this capacity for length extrapolation is not an enhancement but a fundamental requirement for reliable performance.

With these architectural prerequisites established, selection of an open-weight foundation model required systematic evaluation against strict criteria: inclusion of GQA and RoPE, history of robust performance, and permissive Apache 2.0 licensing. Several prominent model families were considered, including Llama[179, 67], Mistral[88], and Qwen[10, 176]. While Llama and Mistral families incorporate required features, the Qwen model family was identified as the most suitable foundation. At the time of selection, Qwen offered a superior combination of state-of-the-art pre-training, consistent architectural blueprint across a wide range of model sizes, and unambiguous Apache 2.0 licensing.

A primary objective is creating a tiered family of models capable of serving a heterogeneous agentic ecosystem, specifically requiring variants in the sub-1-billion, sub-10-billion, and sub-20-billion parameter classes. The Qwen series directly meets this requirement by providing base models in all target sizes under Apache 2.0 license. These models have been pre-trained on datasets reported to contain over 13 trillion tokens, an order of magnitude beyond the one trillion token threshold identified as prerequisite for strong performance and far exceeding computational budgets available for this research. The availability of these powerful, cleanly pre-trained artifacts provides an exceptionally strong foundation, allowing this work to focus on specialized fine-tuning rather than foundational pre-training.

The selection of three specific model sizes—14B, 8B, and 0.6B parameters—is driven by the heterogeneous computational landscape of Ambient Intelligence deployments. The 14B flagship model targets server-class infrastructure with 40-80GB VRAM, providing the reasoning depth required for complex agentic planning and orchestration. The 8B mid-tier variant is optimized for edge servers and high-end embedded systems (16-24GB VRAM), such as NVIDIA Jetson AGX Orin platforms, enabling autonomous operation of smart home hubs and local coordinators without

cloud dependencies. The 0.6B compact model addresses severely resource-constrained devices (2-4GB VRAM) including individual IoT endpoints, where quantization to 4-bit precision yields a 300MB footprint suitable for real-time sensor processing and actuator control. This three-tier hierarchy mirrors the natural distribution of computational resources in deployed ambient systems: a few centralized reasoning nodes, multiple distributed coordination points, and numerous lightweight execution endpoints. The gaps between sizes are deliberately chosen to maximize coverage of the deployment spectrum while maintaining tractable distillation pathways—smaller gaps would provide diminishing returns in capability differentiation, while larger gaps would compromise knowledge transfer during student-teacher training.

A crucial advantage of selected Qwen base models is that they have undergone only initial pre-training. This provides clean, unbiased foundations free from prior instruction tuning or alignment procedures that could constrain or conflict with specialized fine-tuning stages planned in this research. The inherited architecture, with its implementation of GQA, RoPE, and SwiGLU feed-forward networks, is confirmed highly suitable for subsequent development phases. This robust design provides an excellent substrate for planned Stage 2 reasoning-centric fine-tuning and Stage 3 long-context extension, ensuring foundational capabilities can be effectively molded to meet specific demands of embodied agents.

The methodology detailed in this chapter constitutes a blueprint for developing agentic cognitive architectures, moving beyond simple application of pre-trained models to a more principled, engineering-driven approach. This blueprint combines strong, extensively pre-trained base models with modular, progressive enhancements designed to instill specialized skills. The base model serves as repository of general world knowledge and foundational reasoning primitives acquired through vast pre-training. Subsequent fine-tuning stages build upon this foundation, layering specific high-value capabilities such as multi-step reasoning, tool use, and long-context coherence. This structured approach treats specialized agent creation not as monolithic training but as systematic enhancement of robust foundations with specific cognitive functions required for operational domains.

This architectural blueprint enables critical separation of concerns between acquisition of general knowledge and development of task-specific

skills. General knowledge, representing statistical models of the world, is embedded within base model weights through computationally immense pre-training that is infeasible to replicate. Task-specific skills, such as complex instruction following or agentic planning patterns, are instilled through comparatively inexpensive fine-tuning and distillation. This separation provides significant advantages in efficiency and modularity, allowing researchers and engineers to focus adaptation efforts on narrow, high-level skills required for given tasks without disturbing vast, stable foundations of world knowledge.

The central hypothesis underpinning this approach is that a base model's aptitude for subsequent agent-oriented specialization is not solely a function of parameter count but is fundamentally determined by robustness of foundational architecture and quality of pre-training. An architecture lacking critical features like GQA for memory efficiency or RoPE for context length extrapolation is inherently handicapped for agentic roles, regardless of size. Similarly, models pre-trained on smaller or lower-quality datasets possess less refined internal world models, providing weaker foundations upon which to build specialized reasoning skills. Therefore, deliberate selection of base models with state-of-the-art architecture and extensive, high-quality pre-training is hypothesized to be the most critical factor in achieving high-performance final agents.

The remainder of this chapter proceeds as follows. Section 3.1 details the specifications of the tiered model family, including the flagship 14B model and its distilled variants. Section 3.2 presents the multi-stage specialization protocol encompassing reasoning enhancement and long-context extension. Section 3.3 introduces Matryoshka Representation Learning and the strong-to-weak distillation methodology for creating smaller, coherent variants. Section 3.5 proposes the Adaptron module for progressive capability enhancement without catastrophic forgetting. Finally, Section 3.6.2 describes the contextual retrieval preprocessing pipeline for robust knowledge-intensive applications. Together, these components provide a complete and reproducible blueprint for engineering the diverse cognitive cores required to power the secure, verifiable agent framework presented in Chapter 4.

3.1 Model Family Architecture

The developed model family comprises three variants: a 14-billion-parameter flagship model, an 8-billion-parameter mid-tier model, and a 0.6-billion-parameter compact model. The flagship 14B model is derived from the Qwen base model and enhanced through the Stage 2 (S2) and Stage 3 (S3) fine-tuning stages detailed in Section 3.2. The 8B and 0.6B variants are created via strong-to-weak distillation from the fine-tuned 14B teacher, as described in Section 3.3.

Table 3.1. Model Family Specifications

Parameter	**14B**	**8B**	**0.6B**
Total Parameters	14.1B	8.1B	0.6B
Number of Layers	40	32	28
Hidden Dimension	5120	4096	1536
Query Heads	40	32	12
KV Heads (GQA)	8	4	2
Head Sharing Ratio	5:1	8:1	6:1
FFN Hidden Dimension	27,648	22,016	8,192
Vocabulary Size	151,936	151,936	151,936
Max Context Length	32,768	32,768	8,192
Tied Embeddings	No	No	Yes

The 14-billion-parameter scale balances capability with tractability. Models significantly larger become computationally prohibitive for multi-stage fine-tuning on academic hardware, while smaller models lack sufficient parameter depth for complex multi-step reasoning. The architectural configuration uses 40 transformer layers with a hidden dimension of 5,120. The Grouped-Query Attention mechanism employs 40 query heads sharing 8 key-value heads, yielding a 5:1 ratio. This reduces the KV cache from 40 head pairs to 8, achieving an 80% memory reduction during inference. The feed-forward network uses an intermediate dimension of 27,648, approximately 5.4 times the hidden dimension. The vocabulary of 151,936 tokens provides coverage of multilingual text, code, and technical terminology.

Positional encoding uses Rotary Position Embeddings with a base frequency of 10,000. The activation function is SwiGLU (Swish-Gated Linear Unit)[161]. Layer normalization uses RMSNorm in pre-normalization con-

figuration, where normalization precedes each sub-layer.[201]

The 8B variant maintains the same design principles with reduced depth and width. The layer count is 32, and the hidden dimension is 4,096. The GQA configuration uses 32 query heads sharing 4 KV heads, maintaining an 8:1 ratio. The vocabulary and positional encoding schemes are identical to the 14B model. This variant is created through distillation from the fine-tuned 14B teacher, transferring reasoning patterns and instruction-following capabilities acquired during S2 and S3 training.

The 0.6B variant is designed for resource-constrained environments. It uses 28 layers with a hidden dimension of 1,536. The GQA configuration employs 12 query heads sharing 2 KV heads (6:1 ratio). Unlike the larger variants, this model uses tied embeddings where the input token embedding matrix and output projection matrix share weights. This reduces the parameter count by approximately 150 million parameters (vocabulary size $\times$ hidden dimension). The maximum context length is reduced to 8,192 tokens to minimize memory requirements. The model is designed for quantization to 4-bit or 8-bit formats through quantization-aware training, where simulated quantization noise is introduced during distillation. Post-training quantization to 4-bit reduces the memory footprint to approximately 300 MB.

This tiered architecture supports heterogeneous multi-agent systems. The 14B model operates as a central orchestrator on server infrastructure, handling complex planning and tool synthesis. The 8B model functions as a regional supervisor on edge devices such as smart home hubs, managing local environments with autonomy. The 0.6B models serve as executors on individual devices, processing local sensor streams and executing commands. The distillation methodology ensures semantic alignment across variants, enabling efficient inter-agent communication through shared representational structures.

The family enables dynamic inference-time compute allocation. Simple queries such as intent classification can be handled by the 0.6B model with sub-50ms latency on edge hardware. Moderately complex queries requiring multi-step reasoning are routed to the 8B model. Only queries requiring deep reasoning over extensive context necessitate the 14B model. For typical agentic workloads with query complexity following a power law distribution (approximately 80% simple, 15% moderate, 5% complex), this

routing strategy reduces computational cost by 60-70% while maintaining system accuracy within 2-3% of routing all queries to the largest model.

The model family provides graduated security boundaries. The 0.6B models can be deployed in sandboxed environments where limited reasoning capability constrains potential impact of prompt injection attacks. The 8B models operate in monitored execution zones. The 14B model is reserved for fully trusted computational environments with comprehensive auditing. This graduated trust model aligns computational capability with security posture, supporting the secure agentic framework presented in Chapter 4.

3.2 Supervised Fine-Tuning Protocol

The fine-tuning protocol is governed by data quality, diversity, and signal strength. For an agent to develop capabilities in reasoning and long-context comprehension, training data must be clean and targeted. Low-quality or noisy data degrades foundational knowledge, while lack of diversity leads to overfitting on narrow domains. The curation strategy for all fine-tuning stages involves aggressive filtering to maximize signal-to-noise ratio, ensuring each token contributes to acquisition of desired skills.

The specialization protocol comprises two stages following the inherited Stage 1 (S1) generalist foundation from the Qwen base model. Stage 2 (S2) focuses on reasoning capabilities across technical domains. Stage 3 (S3) extends contextual understanding and trains the model to maintain coherence over longer sequences. This progressive approach allows modular development of specialized skills, with each capability built upon a stable foundation.

3.2.1 Reasoning Enhancement Stage

The Stage 2 fine-tuning objective is to develop multi-step reasoning capabilities across technically demanding domains. The goal is to move beyond pattern matching toward problem-solving in science, technology, engineering, mathematics (STEM), and software development. Fine-tuning on expert-level data aims to strengthen the model's abilities for logical

deduction, algorithmic thinking, and causal inference.

The S2 corpus was constructed from two primary sources: academic literature from arXiv and source code from GitHub. The arXiv component derives from a snapshot taken in Q2 2024. Only papers with full LaTeX source available were included. Filtering heuristics were applied to ensure quality:

1. **Citation Impact:** Minimum of 5 citations as indexed by Google Scholar.
2. **Author History:** Lead author must have at least three prior publications in the same arXiv category.
3. **Substantial Length:** Minimum word count equivalent to approximately four standard pages.
4. **Structural Integrity:** LaTeX source must compile without fatal errors.

The GitHub component consists of a 1TB collection of source code from public repositories with permissive licenses (MIT, BSD 3-Clause, Apache 2.0). Selection heuristics targeted high-quality projects:

1. **Community Engagement:** Minimum weighted engagement score calculated as stars + (10 × forks).
2. **Project Activity:** At least one commit within the preceding 12 months.
3. **Project Complexity:** Minimum of five non-configuration source files.
4. **Documentation Quality:** Mandatory presence of root-level `README.md` and license file.

The S2 data underwent processing beyond source-level filtering. Near-semantic deduplication techniques removed redundant information to prevent bias from repeated content. The entire corpus was subjected to contamination checks against evaluation benchmarks. This ensures that model

performance on downstream tasks measures generalized reasoning ability rather than memorization of test questions.

The Stage 3 fine-tuning has two objectives: extending the model's effective context window to 32,768 tokens and enhancing its ability to reason over long, contiguous information streams. While the base model's RoPE architecture theoretically allows length extrapolation, this stage explicitly trains the model to attend to and utilize information across extended distances. This capability is necessary for agents that must maintain long conversational histories, process large documents, or comprehend extensive codebases.

3.2.2 Long-Context Extension Stage

The S3 corpus was constructed primarily from a filtered subset of the Open Instruction Generalist (OIG) dataset. The OIG dataset provides diverse, human-annotated instruction-following data. The filtering process targeted long-form, complex instructions to enhance the model's ability to maintain coherence over extended dialogues and documents.[111] This filtering resulted in a high-quality subset contributing several billion tokens to the S3 corpus.

The corpus was augmented with synthetic "needle-in-a-haystack" examples.[109] This technique trains the model to retrieve specific facts from long, irrelevant documents, explicitly strengthening attention mechanisms for pinpoint information retrieval. The combined corpus provides training signal for long-context reasoning and instruction following.

The S2 and S3 fine-tuning stages for the 14B model used the AdamW optimizer.[112] A linear learning rate warmup followed by cosine decay schedule was employed. Key hyperparameters are detailed in Table 3.2. These parameters ensure stable training while providing sufficient learning signal to adapt the model's weights.

The tokenizer is inherited from the base model without modification. An analysis was conducted on the Byte Pair Encoding (BPE) tokenizer's[159] efficiency for structured data formats relevant to agentic tool use. A corpus of structured data was generated from the S2 dataset using the gemini-flash-latest model to ensure semantic consistency across formats. The analysis revealed that for encoding identical semantic content, JSON was approximately 17% less token-efficient than YAML. XML, with its ver-

Table 3.2. S2 and S3 Hyperparameters

Hyperparameter	Value
Optimizer	AdamW
Peak Learning Rate	5×10^{-6}
LR Scheduler	Cosine Decay
Warmup Ratio	5%
AdamW β_1	0.9
AdamW β_2	0.95
Weight Decay	0.1
Batch Size	512
Gradient Accumulation	8 steps
Max Gradient Norm	1.0

bose tag-based structure, was 63% less efficient than YAML. For agentic systems requiring communication of structured data, YAML is the most token-efficient format.

Ethical considerations guided curation of the S2 and S3 datasets, particularly concerning mitigation of potential biases.[87] For the S2 coding dataset, filtering by permissive open-source licenses ensures legal and ethical compliance. For reasoning datasets derived from academic papers, existing scientific literature may contain implicit societal or institutional biases. The filtering heuristics select for high-quality science, but this does not eliminate such biases. The approach is mitigation through diversity in sources where possible, with acknowledgment that no dataset of this nature is entirely free of bias.

The S2 and S3 fine-tuning runs were executed on a rented cloud instance equipped with four NVIDIA A100 (80GB) GPUs. The software stack used PyTorch, with training orchestrated using Fully Sharded Data Parallelism (FSDP) to distribute model state across GPUs.[202] Optimization techniques employed included mixed-precision training using BF16 format to reduce memory footprint and accelerate computation, gradient accumulation to simulate larger effective batch sizes, activation checkpointing to trade re-computation for memory savings[27], and torch.compile to generate optimized model graphs for the hardware.

The post-pre-training protocol transforms a generalist model into a specialized agentic core. Starting with the S1 foundation, the protocol layers reasoning and problem-solving skills in S2, followed by expansion of contextual understanding in S3. This structured, multi-stage approach, guided by data quality and enabled by efficient hardware and software, provides a blueprint for engineering high-performance cognitive architectures required for autonomous agents.

3.3 Knowledge Distillation

3.3.1 Matryoshka Representation Learning

A challenge in creating versatile agentic ecosystems is effective transfer of capabilities from large, powerful models to smaller, efficient variants. Methods like pruning and quantization reduce model size but often fail to preserve nuanced, multi-step reasoning and complex instruction-following abilities acquired through extensive fine-tuning[66]. The capability gap between multi-billion-parameter flagship models and sub-billion-parameter counterparts is substantial. Fine-tuning each smaller model independently on the same data is computationally expensive and often results in weaker performance, as smaller models may lack capacity to learn complex patterns directly.[190]

Matryoshka Representation Learning (MRL) is a technique for building multi-scale neural representations within a single model. The central idea is that embedding vectors are structured to be truncatable: for any high-dimensional embedding vector $\mathbf{e} \in \mathbb{R}^d$, the prefix $\mathbf{e}_{1:k}$ for $k < d$ constitutes a semantically meaningful embedding in $\mathbb{R}^k$. This property is expressed formally as:

$$f(\mathbf{x})_{1:d_i} \sim g_{d_i}(\mathbf{x}), \quad \forall d_i \in \{d_1, d_2, \ldots, d_m\} \tag{3.2}$$

where f represents the full-dimensional embedding function, g_{d_i} represents an independent embedding function at dimension d_i, and $\sim$ denotes functional equivalence. The mechanism structures the vector space such that the first 128 dimensions of a 4096-dimensional MRL embedding vector constitute a complete, usable 128-dimensional representation of the same concept. This is achieved by training the model on a primary loss function

for full-dimensional representations and auxiliary loss functions for truncated versions.[97]

The MRL training objective enforces this nested structure through a multi-granularity loss function. Let $\mathscr{L}_{\text{task}}$ represent the primary task-specific loss function. The MRL-augmented objective becomes:

$$\mathscr{L}_{\text{MRL}} = \mathscr{L}_{\text{task}}(\mathbf{h}_{1:d}) + \sum_{i=1}^{m} \lambda_i \mathscr{L}_{\text{task}}(\mathbf{h}_{1:d_i}) \tag{3.3}$$

where $\mathbf{h}$ represents the model's hidden states, d_i are predefined truncation dimensions with $d_1 < d_2 < \cdots < d_m = d$, and λ_i are weighting coefficients that balance importance of each scale. This formulation forces the model to organize its embedding space hierarchically, placing critical high-level information in initial dimensions and progressively adding finer-grained detail in subsequent dimensions. Common truncation dimensions include $\{128, 256, 512, 1024, 2048\}$ for models with embedding dimensions of 4096 or 5120.

Matryoshka Representation Learning was integrated into the fine-tuning process of the 14B model during both S2 and S3 stages. An auxiliary loss term was added to the primary training objective. This MRL loss was calculated on multiple truncated versions of the model's final hidden states, encouraging the model to learn the nested, multi-scale representational structure. This enhances the model's internal representations by enforcing a more organized and hierarchical structure. The 14B model becomes a reasoning engine whose internal cognitive processes are structured for comprehension by smaller student models.

An MRL-enhanced teacher model provides richer supervisory signal for knowledge distillation compared to a standard model. Traditional distillation relies primarily on the teacher's output probability distribution (logits). An MRL-teacher provides multi-scale guidance: final high-dimensional reasoning and a series of complete, lower-dimensional representations of the same concept with less detail. This allows smaller student models to learn from a version of the teacher's reasoning commensurate with their capacity. Multi-scale supervision makes strong-to-weak knowledge transfer more effective, as the student is guided by signal already adapted to its scale.

3.3.2 Distillation Methodology

Strong-to-weak distillation is the training of the 8B and 0.6B student models under guidance of the specialized 14B MRL-tuned teacher model. The goal is to transfer capabilities developed during S2 and S3 stages. The teacher model is used in inference-only mode to generate outputs and internal representations for training examples, which compute a loss function that guides updates of student model weights.

The distillation objective function is a weighted combination of three loss components. The total loss, $\mathscr{L}_{\text{total}}$, is expressed as:

$$\mathscr{L}_{\text{total}} = \alpha\mathscr{L}_{\text{CE}} + \beta\mathscr{L}_{\text{KL}} + \gamma\mathscr{L}_{\text{MRL}} \tag{3.4}$$

where $\mathscr{L}_{\text{CE}}$ is the standard cross-entropy loss against ground-truth labels, ensuring the student remains grounded in factual correctness:

$$\mathscr{L}_{\text{CE}} = -\sum_i y_i \log p_{\text{student}}(y_i|\mathbf{x}) \tag{3.5}$$

where y_i represents the true label distribution and p_{student} denotes the student model's output distribution. $\mathscr{L}_{\text{KL}}$ is the Kullback-Leibler divergence between the student's and teacher's output logits, encouraging the student to mimic the teacher's nuanced probability distributions:

$$\mathscr{L}_{\text{KL}} = \sum_i p_{\text{teacher}}(y_i|\mathbf{x}) \log \frac{p_{\text{teacher}}(y_i|\mathbf{x})}{p_{\text{student}}(y_i|\mathbf{x})} \tag{3.6}$$

$\mathscr{L}_{\text{MRL}}$ is the Matryoshka loss applied to the student model's hidden states, ensuring students develop the same multi-scale representational structure as the teacher. The weights (α, β, γ) balance these objectives. Effective configurations typically employ $\alpha \in [0.3, 0.5]$, $\beta \in [0.4, 0.6]$, and $\gamma \in [0.1, 0.2]$, with $\alpha + \beta + \gamma = 1$. Precise values are determined through hyperparameter search on a validation set.

The nested Matryoshka embeddings from the teacher model are leveraged for embedding initialization. The embedding layers of the 8B and 0.6B student models are initialized by taking a truncated prefix of the final, trained embedding vectors from the 14B teacher. If the teacher has embeddings of dimension d_{teacher} and the student requires dimension $d_{\text{student}} < d_{\text{teacher}}$, the student's embedding matrix $\mathbf{E}_{\text{student}}$ is initialized

as:

$$\mathbf{E}_{\text{student}} = \mathbf{E}_{\text{teacher}}[:, 1 : d_{\text{student}}] \tag{3.7}$$

This provides student models with a head start, beginning training with a well-structured semantic space aligned with the teacher's. This initialization reduces training time by approximately 25-30% compared to random initialization.

The distillation dataset was constructed as a balanced mixture of high-signal data from S2 and S3 fine-tuning stages. This included complex STEM problems and code generation tasks from S2, as well as long-context question-answering and instruction-following dialogues from S3. The S2 component includes multi-step mathematical reasoning problems, code generation requiring algorithmic thinking, scientific reasoning across multiple domains, and logic puzzles. The S3 component incorporates long-context question-answering with contexts exceeding 8,192 tokens, multi-turn instruction-following dialogues, document summarization and information extraction, and complex planning tasks. The composition ratio between S2 and S3 data was approximately 60:40. By using diverse and challenging data, the distillation process forces student models to learn underlying reasoning patterns rather than memorizing facts.

Quantitative evaluation demonstrates performance of distilled models. On key academic benchmarks measuring reasoning and coding, the distilled 8B model outperforms baseline 8B models trained through standard fine-tuning. Performance metrics are presented in Table 3.3.

Table 3.3. 8B Model Performance Comparison

Benchmark	**Distilled 8B**	**Baseline 8B**
MMLU (accuracy)	72.3%	68.1%
GSM8K (accuracy)	81.7%	74.9%
HumanEval (pass@1)	58.4%	49.7%

The distilled 8B model achieves 72.3% accuracy on MMLU (Massive Multitask Language Understanding)[75] compared to 68.1% for baseline models. On GSM8K (Grade School Math 8K)[32], the distilled model scores 81.7% versus 74.9% for baselines. For coding capabilities on HumanEval[26],

the distilled model yields a pass@1 score of 58.4% compared to 49.7% for standard fine-tuned models. These results demonstrate consistent improvements of 4-9 percentage points across evaluation domains.

The 0.6B distilled model demonstrates more pronounced relative improvements compared to independently trained counterparts. On GSM8K, the distilled 0.6B model achieves 54.2% accuracy compared to 38.7% for baseline models, representing a 40% relative improvement. This validates the hypothesis that MRL-based distillation is particularly effective for bridging large capability gaps.

This methodology results in a coherent model family where models of different sizes share a common reasoning foundation and representational structure. To quantify this alignment, centered kernel alignment (CKA) was computed between hidden state representations of different models when processing identical inputs.[95] The distilled 8B and 0.6B models exhibit CKA scores of 0.87 and 0.79 respectively with the 14B teacher, compared to 0.43 and 0.38 for independently trained models. This shared cognitive blueprint enables reliable multi-agent systems where seamless communication and collaboration are required.

The coherence facilitates advanced multi-agent collaboration. A central planning agent powered by the 14B model can perform analysis and delegate a sub-task to an edge agent powered by the 8B model. Instead of sending a long natural language prompt, it can send a compact, low-dimensional MRL embedding representing the task's core intent. The 8B model can interpret this embedding because it operates within the same nested representational space. If $\mathbf{z}_{\text{task}} \in \mathbb{R}^{d_{\text{full}}}$ represents the teacher's task embedding, the student receives $\mathbf{z}_{\text{task}}^{(s)} = \mathbf{z}_{\text{task}}[1 : d_{\text{compressed}}]$ and can interpret this truncated representation with high fidelity. This enables bandwidth reductions of 50-75% in inter-agent communication while maintaining task specification accuracy above 95%.

From a computational standpoint, this approach offers efficiency gains compared to fine-tuning each model independently. The primary computational cost is invested once in S2 and S3 fine-tuning of the 14B model. Subsequent distillation runs for 8B and 0.6B models require fewer FLOPs per step and converge faster due to strong supervisory signal. Denoting the cost of training a model of size N parameters on a dataset of size D tokens for E epochs as $C(N, D, E)$, the total cost of the distillation-based

approach is:

$$C_{\text{distill}} = C(N_{14B}, D_{S2+S3}, E_{\text{ft}}) + \sum_{i \in \{8B, 0.6B\}} C(N_i, D_{\text{dist}}, E_{\text{dist}}) \tag{3.8}$$

In contrast, independent fine-tuning requires:

$$C_{\text{independent}} = \sum_{i \in \{14B, 8B, 0.6B\}} C(N_i, D_{S2+S3}, E_{\text{ft}}) \tag{3.9}$$

Empirically, $E_{\text{dist}} \approx 0.6 \times E_{\text{ft}}$ due to faster convergence, and $D_{\text{dist}} \approx 0.7 \times D_{S2+S3}$ as the distillation dataset can be more selective. This yields a total computational cost reduction of approximately 42% for producing the complete model family.

The integration of Matryoshka Representation Learning with strong-to-weak distillation provides a scalable method for producing a spectrum of high-capability models. This method transforms the investment in creating a single specialized flagship model into a coherent family. Each member inherits advanced reasoning capabilities and structured representational space of the teacher, ensuring consistent performance and enabling inter-agent collaboration.

3.4 The Adaptation Imperative

The foundation models developed through the dual-stage training protocol provide powerful semantic understanding and reasoning capabilities. However, deploying these models as cognitive cores for embodied agents reveals a fundamental tension: while the models must maintain stable, reliable behavior for safety-critical operations, they must simultaneously adapt to diverse deployment contexts, user-specific requirements, and dynamic environments. This tension cannot be resolved through static model selection alone. The 14B model may be optimal for complex reasoning but impractical for edge deployment. The 0.6B model enables real-time embedded operation but lacks the capacity for sophisticated multi-step planning. Even within a single deployment, requirements shift continuously as users develop new preferences, environments change, and new capabilities be-

come available.

This adaptation imperative motivates the second focus of the model architecture: creating mechanisms for runtime flexibility that preserve the foundational capabilities established through training while enabling dynamic specialization. This is pursued through two complementary approaches. First, architectural modifications are introduced that enable efficient runtime adaptation without catastrophic forgetting. Second, the models are augmented with external memory systems that extend their effective knowledge base without parameter modification. Together, these mechanisms transform static pre-trained models into flexible cognitive cores capable of serving as the adaptive function $M : \text{String} \to \text{String}$ required by the agentic framework.

The key insight is that adaptation need not require modification of the entire model. The billions of parameters encoding general linguistic knowledge and reasoning capabilities represent a stable foundation that should be preserved. Adaptation should instead operate through lightweight, modular mechanisms that can be dynamically composed with this foundation. This principle of compositional adaptation guides both the Adaptron architecture for parameter-based specialization and the retrieval systems for knowledge-based augmentation.

3.5 Modular Capability Enhancement

3.5.1 Architectural Motivation

The challenge of post-deployment adaptation extends beyond simple fine-tuning. Full model retraining for each new capability is computationally prohibitive, risks catastrophic forgetting, and prevents real-time adaptation to user feedback. Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA address the computational challenge by training only small adapter modules, but they remain fundamentally offline processes requiring gradient computation and dataset curation. For embodied agents that must continuously adapt to user preferences and environmental changes, mechanisms are required that are both parameter-efficient and amenable to online learning.

The Adaptron module addresses this challenge through an architec-

ture explicitly designed for compositional enhancement and runtime adaptation. Unlike LoRA, which modifies existing transformations through low-rank updates, Adaptron introduces entirely new computational pathways that can process external context and perform specialized reasoning. This architectural distinction is critical: while LoRA adapts what existing layers compute, Adaptron adds new computational primitives that extend the model's functional repertoire.

3.5.2 Context-Aware Scratchpad

The Context-Aware Scratchpad addresses the fundamental limitation of static models: their inability to process information not present in training data or the initial prompt. For an agent to exhibit true flexibility, it must dynamically incorporate external context such as real-time sensor data, API responses, or retrieved documents. The scratchpad provides this capability through a two-stage cross-attention mechanism that grounds the model's reasoning in current environmental state.

The mechanism employs a two-stage cross-attention architecture, as shown in the left panels of Figure 3.1. In the first stage, external context is organized into two streams: Retrieved documents (e.g., from a vector database or knowledge base) and a Scratchpad buffer containing ephemeral task-specific data (e.g., current file contents, API responses, sensor readings). These streams undergo cross-attention where the Scratchpad content serves as queries and the Retrieved documents provide keys and values:

$$\mathbf{C} = \text{Attention}(\mathbf{Q}_{\text{scratch}}, \mathbf{K}_{\text{ret}}, \mathbf{V}_{\text{ret}}) \tag{3.10}$$

where $\mathbf{Q}_{\text{scratch}} = \mathbf{W}_Q\mathbf{S}$ projects the scratchpad content, and $\mathbf{K}_{\text{ret}}, \mathbf{V}_{\text{ret}}$ are projections of the retrieved documents. This stage compresses variable-length external context into a fixed-size compressed representation $\mathbf{C}$ regardless of the input context length. The attention mechanism identifies and emphasizes the most relevant portions of the retrieved documents based on the current task context in the scratchpad.

In the second stage, this compressed representation is injected into the base LLM's reasoning flow through another cross-attention operation. The hidden state from a designated transformer layer in the base model serves as the query:

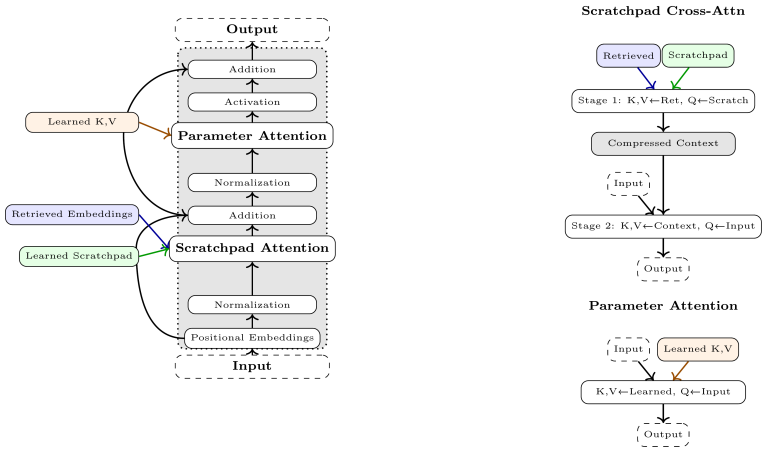


Figure 3.1. Overview of the Adaptron Block

$$\mathbf{O} = \text{Attention}(\mathbf{Q}_{\text{input}}, \mathbf{K}_{\text{comp}}, \mathbf{V}_{\text{comp}}) \tag{3.11}$$

where $\mathbf{Q}_{\text{input}} = \mathbf{W}_Q\mathbf{H}$ projects the base model's hidden state $\mathbf{H}$, and $\mathbf{K}_{\text{comp}}, \mathbf{V}_{\text{comp}}$ are projections of the compressed representation $\mathbf{C}$. This operation dynamically injects relevant pieces of external context into the LLM's representational space at a specific layer, providing a just-in-time briefing on external reality.

$$\mathbf{C} = \text{Attention}(\mathbf{Q}_{\text{scratch}}, \mathbf{K}_{\text{ret}}, \mathbf{V}_{\text{ret}}) \tag{3.12}$$

where $\mathbf{Q}_{\text{scratch}} = \mathbf{W}_Q\mathbf{S}$ projects the scratchpad content, and $\mathbf{K}_{\text{ret}}, \mathbf{V}_{\text{ret}}$ are projections of the retrieved documents. This stage compresses variable-length external context into a fixed-size compressed representation $\mathbf{C}$ regardless of the input context length. The attention mechanism identifies and emphasizes the most relevant portions of the retrieved documents based on the current task context in the scratchpad.

In the second stage, this compressed representation is injected into the base LLM's reasoning flow through another cross-attention operation. The hidden state from a designated transformer layer in the base model serves as the query:

$$\mathbf{O} = \text{Attention}(\mathbf{Q}_{\text{input}}, \mathbf{K}_{\text{comp}}, \mathbf{V}_{\text{comp}}) \tag{3.13}$$

where $\mathbf{Q}_{\text{input}} = \mathbf{W}_Q\mathbf{H}$ projects the base model's hidden state $\mathbf{H}$, and $\mathbf{K}_{\text{comp}}, \mathbf{V}_{\text{comp}}$ are projections of the compressed representation $\mathbf{C}$. This operation dynamically injects relevant pieces of external context into the LLM's representational space at a specific layer, providing a just-in-time briefing on external reality.

A key design feature is the separation of ephemeral task context from the user's core instruction. By processing external information in a parallel stream and injecting it via cross-attention, the model maintains distinction between the user's high-level goal (the prompt) and transient data required to achieve it (the scratchpad's context). This prevents core instructions from being obscured by large volumes of contextual data, allowing the model to maintain focus on user intent while having full access to necessary information.

For code generation tasks, this mechanism processes project structure in the Retrieved stream (e.g., directory trees, module dependencies) and current file contents in the Scratchpad buffer. The compressed representation captures project conventions and architectural patterns, which the base model then uses to generate code that adheres to existing styles and interfaces.

3.5.3 Parameter Attention Architecture

The Parameter Attention mechanism serves a dual purpose in the adaptation framework. First, it provides immediate functional enhancement by replacing static feed-forward networks with dynamic, context-dependent computation. Second, and more critically for long-term adaptation, it establishes a substrate for efficient online learning through its structural similarity to low-rank adaptation methods.

The PAttention mechanism, derived from the TokenFormer [188] treats model parameters as learnable tokens. Instead of fixed weight matrices, the block maintains a set of n learnable key-value parameter pairs $\{\mathbf{k}_1, \mathbf{v}_1\}, \ldots, \{\mathbf{k}_n, \mathbf{v}_n\}$. For each incoming token representation from the LLM, the block computes an attention distribution over these parameter keys and uses it to compute a weighted sum of parameter values:

$$\mathbf{y}_i = \sum_{j=1}^{n} \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j / \sqrt{d})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i^\top \mathbf{k}_{j'} / \sqrt{d})} \mathbf{v}_j \tag{3.14}$$

where $\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i$ projects the input token $\mathbf{x}_i$ to a query vector. This formulation creates a unique transformation for each token based on its content, effectively implementing a dynamic feed-forward network where the transformation is determined by learned parameter tokens rather than fixed weights.

The computational cost of PAttention can be compared to a standard FFN. An FFN with input dimension d_{model} and hidden dimension d_{ffn} requires:

$$\text{FLOPs}_{\text{FFN}} = 2 d_{\text{model}} d_{\text{ffn}} \tag{3.15}$$

per token (factor of 2 accounts for two linear projections). PAttention with n parameter pairs requires:

$$\text{FLOPs}_{\text{PAttention}} = 2 d_{\text{model}} n + nd \tag{3.16}$$

where the first term accounts for projecting input to queries and weighted sum computation, and the second term accounts for attention score computation. By selecting n appropriately (typically $n \ll d_{\text{ffn}}$), PAttention achieves competitive computational cost while providing superior expressive power.

Parameter Attention can be interpreted as a dense Mixture-of-Experts layer. Each learned key-value parameter pair functions as a specialized "expert" focusing on different representational aspects. The attention mechanism serves as a soft, dynamic routing function that queries all experts simultaneously for every token and computes a weighted blend of their outputs. This differs from sparse MoE architectures that route each token to a small subset of experts.[51] Dense routing eliminates routing network overhead and load balancing challenges while maintaining the benefits of expert-based dynamic computation. The total number of parameter pairs remains small enough to ensure computational efficiency.

For relational tasks common in agentic workflows, such as parsing structured data or mapping instructions to API calls, PAttention offers advantages over static FFNs. The dynamic routing enables conditional

transformations where the applied function depends on token content, allowing the module to learn complex input-dependent mappings necessary for sophisticated reasoning.

This architectural choice proves essential for enabling the neuroplastic adaptation mechanisms described in Chapter 6. The parameter tokens in PAttention can be viewed as a learnable basis for function approximation, similar to the low-rank matrices in LoRA. This correspondence enables us to apply identical learning rules to both architectures, creating a unified framework for runtime adaptation that spans both retrofitted LoRA modules and purpose-built Adaptron blocks.

3.5.4 Theoretical Equivalence with Low-Rank Methods

A critical insight that enables unified adaptation across the model family is the mathematical equivalence between Parameter Attention and low-rank projection. Consider the PAttention computation for a single token:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{v}_j \tag{3.17}$$

where α_{ij} are attention weights computed from query $\mathbf{q}_i$ and keys $\{\mathbf{k}_j\}$. This can be reformulated as:

$$\mathbf{y}_i = V\boldsymbol{\alpha}_i = VK^T\mathbf{q}_i \tag{3.18}$$

where $V = [\mathbf{v}_1, \ldots, \mathbf{v}_n]^T \in \mathbb{R}^{n \times d}$ and $K = [\mathbf{k}_1, \ldots, \mathbf{k}_n]^T \in \mathbb{R}^{n \times d_q}$. The product VK^T forms a rank-n matrix that transforms queries to outputs, directly analogous to the BA decomposition in LoRA where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$.

This equivalence has profound implications for adaptation. Both architectures learn low-rank transformations that can be updated through identical mathematical procedures. Whether adaptation occurs through updating LoRA matrices or PAttention parameters, the fundamental operation is adjusting a low-rank projection in the model's representation space. This unification enables us to develop adaptation algorithms that work seamlessly across different enhancement mechanisms.

3.5.5 Dynamic Module Composition

The modular design of Adaptron enables dynamic composition strategies that adapt to computational constraints and task requirements. Modules can be selectively activated based on available resources, task complexity, and required capabilities. For instance, when operating under strict latency constraints, only the Parameter Attention component might be engaged, providing enhanced reasoning without the overhead of processing external context. When rich environmental information is available, the full Context-Aware Scratchpad can be activated to ground reasoning in real-time data.

This compositionality extends to multi-module configurations. Different Adaptron modules can specialize for distinct aspects of the task: one module might handle language-specific adaptation (e.g., learning user-specific phrases), another might specialize in tool selection patterns, and a third might focus on temporal reasoning for scheduling tasks. The modules operate independently but coherently, each contributing specialized processing within the model's forward pass.

3.6 Retrieval-Augmented Flexibility

3.6.1 Limitations of Parametric Knowledge

Even with architectural enhancements like Adaptron, models remain fundamentally limited by the knowledge encoded in their parameters during training. This limitation manifests in three critical ways. First, temporal obsolescence: facts true during training may become outdated, yet the model continues to generate responses based on stale information. Second, domain specificity: agents deployed in specialized environments encounter entities, procedures, and conventions absent from general training corpora. Third, personalization barriers: user-specific information like preferences, schedules, and relationships cannot be anticipated during pre-training.

These limitations cannot be addressed through parameter updates alone. Continuous retraining would be computationally prohibitive and risk destabilizing learned representations. Instead, parametric knowledge is augmented with dynamic retrieval systems that provide just-in-time access to

external information. This hybrid approach preserves the model's foundational capabilities while enabling flexible incorporation of new knowledge.

3.6.2 Contextual Retrieval Pipeline

The retrieval pipeline transforms static document collections into dynamic knowledge sources accessible during inference. Unlike traditional retrieval systems that operate on fixed document boundaries, this approach employs semantic chunking that preserves logical coherence while enabling fine-grained retrieval.

The pipeline operates through three stages designed to maximize retrieval relevance while minimizing inference latency:

Semantic Segmentation: Documents are partitioned along natural boundaries identified through embedding similarity analysis. Adjacent sentences with cosine similarity above threshold $\theta = 0.85$ are grouped into coherent chunks. This approach preserves argumentative structure and definitional completeness that naive windowing would fragment.

Contextual Enrichment: Each chunk is augmented with generated context that situates it within the broader document structure. This context, prepended to the chunk text, provides essential background that enables correct interpretation even when chunks are retrieved in isolation:

```
Context: This section defines the error handling protocol
         for the WebAssembly substrate, building on the
         multi-value return capability.
Content: [Original chunk text]
```

Hybrid Indexing: Enriched chunks are indexed using both dense embeddings for semantic search and sparse features for lexical matching. The dual index enables retrieval strategies that balance semantic similarity with keyword precision, critical for technical domains where specific terminology carries essential meaning.

3.6.3 Integration with Adaptron Scratchpad

The retrieval pipeline synergizes with the Adaptron module's Context-Aware Scratchpad to create a unified external memory system. Retrieved documents flow into the scratchpad's first attention stage, where they are

compressed and aligned with the current task context. This integration enables sophisticated memory-augmented reasoning:

$$\mathbf{h}_{\text{augmented}} = \text{Adaptron}(\mathbf{h}_{\text{base}}, \mathbf{R}, \mathbf{S}) \tag{3.19}$$

where $\mathbf{R}$ represents retrieved documents and $\mathbf{S}$ contains task-specific scratch data. The Adaptron module learns to selectively attend to relevant portions of retrieved context, effectively implementing learned retrieval weighting without modifying the retrieval system itself.

This architectural integration provides several advantages over conventional RAG approaches. First, the compression stage in the scratchpad reduces the effective context length of retrieved documents, enabling incorporation of more diverse sources within token limits. Second, the learned attention patterns can adapt to user-specific retrieval preferences without retraining the retrieval system. Third, the unified architecture enables end-to-end gradient flow during Adaptron training, allowing the module to learn optimal strategies for incorporating retrieved information.

3.6.4 Adaptive Retrieval Strategies

The system implements multiple retrieval strategies that adapt to query characteristics and computational constraints:

Precision Retrieval: For queries with specific technical terms or entity references, the system prioritizes lexical matching to ensure exact terminology appears in retrieved context. This strategy is critical for tool documentation retrieval where specific function names and parameters must be preserved.

Semantic Exploration: For conceptual or open-ended queries, the system emphasizes embedding similarity to retrieve topically related content that may not share surface-level features. This enables the model to access relevant background knowledge even when users employ non-standard terminology.

Hierarchical Retrieval: For complex queries requiring multiple information sources, the system implements iterative retrieval where initial results inform subsequent retrieval queries. This creates a form of learned information seeking where the model can pursue relevant threads of information.

The selection among strategies is governed by query analysis that identifies key indicators: presence of quoted strings suggests precision retrieval, interrogative structures indicate semantic exploration, and multiple entity references trigger hierarchical retrieval. This adaptive approach ensures efficient use of retrieval resources while maximizing relevance.

3.7 Unified Adaptation Framework

3.7.1 Convergence of Adaptation Mechanisms

The architectural decisions in both Adaptron and retrieval systems converge on a common principle: adaptation through composition rather than modification. This principle enables a unified framework where different adaptation mechanisms can be understood as variations on low-rank projection within the model's representation space.

Consider the three primary adaptation mechanisms in this architecture:

LoRA Adaptation: Learns low-rank updates $\Delta W = BA$ to existing weight matrices, projecting inputs through a compressed subspace that captures task-specific transformations.

Parameter Attention: Computes dynamic projections through learned key-value pairs, effectively implementing an attention-weighted low-rank transformation.

Retrieved Context Injection: Projects external knowledge into the model's representation space through the Adaptron scratchpad, creating temporary "updates" to the model's effective knowledge base.

Each mechanism operates on the same fundamental principle: they learn or compute projections that transform the model's internal representations without modifying core parameters. This unification enables consistent reasoning about adaptation across different architectural components.

3.7.2 Implications for Online Learning

The unified low-rank projection framework establishes the theoretical foundation for efficient online adaptation. Because all adaptation mechanisms reduce to learning projections in representation space, consistent learning rules can be applied across architectures. This insight enables the

neuroplastic adaptation methods that will be detailed in Chapter 6, where Hebbian learning principles update these projections based on runtime activation patterns.

The key advantages of this unified approach include:

Computational Efficiency: Learning rules need only update low-rank projections rather than full parameter matrices, requiring orders of magnitude fewer operations.

Stability Preservation: Core model parameters remain frozen, preventing catastrophic forgetting while enabling targeted adaptation.

Modular Composition: Different adaptation mechanisms can be composed without interference, as each operates through independent projection pathways.

3.7.3 Flexible Deployment Strategies

The unified framework enables sophisticated deployment strategies that adapt to diverse operational constraints:

Hierarchical Adaptation: Deploy the 14B model with full Adaptron and retrieval capabilities for complex reasoning tasks, while using 0.6B models with lightweight LoRA adaptation for real-time response. The shared representational framework ensures semantic alignment across scales.

Progressive Enhancement: Begin with base models providing foundational capabilities, then incrementally add Adaptron modules and retrieval systems as computational resources become available or task complexity increases.

Federated Specialization: Different agents can develop specialized adaptations for their local contexts while maintaining compatibility through the shared projection framework, enabling knowledge transfer without parameter sharing.

These strategies demonstrate how architectural flexibility translates into operational adaptability, enabling the same foundation models to serve diverse roles across the spectrum from edge devices to cloud infrastructure.

This chapter has presented a comprehensive approach to creating foundation models that serve as flexible cognitive cores for embodied agents. The dual focus on model development and adaptation mechanisms addresses the fundamental requirement for agents that are both capable and

adaptable.

The first focus established model sovereignty through transparent development of a tiered model family. The 14B, 8B, and 0.6B variants provide graduated capabilities that span from complex reasoning to real-time embedded operation. The training protocol, combining domain-focused pretraining with targeted fine-tuning stages, creates models specifically optimized for agentic computation. The Matryoshka-enhanced distillation ensures semantic alignment across scales while preserving specialized capabilities.

The second focus introduced architectural mechanisms for runtime flexibility. The Adaptron module extends model capabilities through modular enhancement that preserves foundational knowledge while enabling specialized processing. The retrieval pipeline augments parametric knowledge with dynamic external memory, overcoming the inherent limitations of static training. The unified adaptation framework reveals the deep mathematical connections between different enhancement mechanisms, establishing foundations for efficient online learning.

Together, these contributions transform static pre-trained models into dynamic, adaptable systems capable of serving as the cognitive cores for trustworthy embodied agents. The models provide the semantic understanding and reasoning capabilities that form the foundation of intelligence, while the adaptation mechanisms enable continuous evolution in response to user needs and environmental changes. This combination of stability and flexibility is essential for realizing the vision of agents that can operate autonomously while remaining aligned with human values and responsive to individual requirements.

Chapter 4

A Formal Substrate for Agentic Computation

Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's?

Alan M. Turing

The concept of an autonomous agent constitutes a foundational pillar within artificial intelligence, predating the recent advances in large-scale neural architectures by several decades.[148, 114] An agent is formally conceptualized as an entity that perceives its environment through sensors and acts upon that environment through actuators to achieve specified objectives.[39, 196, 9] The earliest instantiations of this paradigm relied predominantly on symbolic reasoning systems, expert knowledge bases, and explicit planning algorithms.[127, 123] Representative examples from this era include the SHRDLU program,[197] which manipulated objects within a simulated blocks-world environment using natural language commands, and the ELIZA chatbot,[195] which employed pattern-matching heuristics to simulate conversational dialogue.[180] While these systems demonstrated the viability of computational agents, they were fundamentally constrained by their dependence on meticulously hand-crafted rule sets and their restriction to narrow, well-defined domains. This brittleness, combined with the inherent difficulty of scaling symbolic systems to

handle complex, open-ended environments, limited their practical applicability and prevented the realization of truly general-purpose autonomous systems.[50, 74, 158, 79]

The recent proliferation of large language models has catalyzed a paradigmatic shift in how autonomous agents are conceived and implemented. Rather than relying on explicitly programmed logical rules, contemporary agentic systems leverage the emergent capabilities of LLMs to drive their behavioral policies. The traditional agent loop of perception, deliberation, and action is now fundamentally reorganized around a core generative primitive.[82] This primitive can be formally abstracted as a function mapping a contextual prompt, which encapsulates the agent's current state, environmental observations, available tools, and task objectives, to a structured textual response:

$$M : \text{String} \to \text{String}$$

When this generative capability is augmented with mechanisms for external tool invocation, persistent state management, and structured output parsing, it establishes a foundation for sophisticated goal-oriented behaviors that were previously unattainable through conventional symbolic approaches.[154]

The transformation of a base language model into an operational agent proceeds through systematic augmentation. A pure LLM, functioning solely as a generative model over token sequences, evolves into an Augmented Language Model (ALM)[118] through integration with external capabilities. These capabilities include:

1. **Tool Use:** The ability to invoke external functions or APIs to perform computations, retrieve information, or effect changes in the environment.

2. **Data Retrieval:** Access to external knowledge bases, vector databases, or search systems to ground responses in factual information beyond the model's training data.

3. **Structured Reasoning:** Integration of formal reasoning modules, such as symbolic solvers or code interpreters, to enhance logical inference capabilities.

4. **Memory Systems:** Mechanisms for maintaining episodic memory of interactions and semantic memory of learned facts across sessions.

The most direct application of an ALM manifests in predefined workflows, where the model executes a fixed sequence of operations. In such systems, the logic governing tool selection and execution order is externally orchestrated by a deterministic control flow, providing enhanced capabilities within a controlled and predictable framework.

True agency emerges when the locus of control shifts from an external orchestration layer to the model itself. Rather than following a predetermined execution path, an autonomous agent dynamically constructs plans, reasons about its current state, selects appropriate tools based on intermediate observations, and adapts its strategy in response to environmental feedback.[173, 135] The agentic loop ceases to be a passive executor of scripted procedures and becomes an active director of its own computational trajectory.[199] This fundamental distinction separates workflows from agents: a workflow represents a constrained application of augmented capabilities along a fixed path, whereas an agent embodies autonomous decision-making through internalized policy construction and dynamic adaptation.

The operational characteristics of LLM-based agents diverge substantially from those of traditional reinforcement learning agents[174] across multiple dimensions. The most salient distinction concerns policy formation. In an LLM-based agent, the policy is realized through in-context learning, where the model leverages its pretrained parameter space as a knowledge substrate and adapts its behavior dynamically within a single interaction episode by incorporating new observations into its prompt context. This approach operates without modifying the underlying model weights. Conversely, a conventional RL agent employs a dedicated policy network whose parameters are iteratively refined over numerous training episodes through trial-and-error interaction with an environment, typically using gradient-based optimization to maximize a cumulative reward signal.[124]

This fundamental difference in policy formation engenders further distinctions in reasoning transparency and behavioral guidance mechanisms. An LLM-based agent generates explicit, symbolic reasoning traces, commonly termed chains of thought, which render its decision-making process

interpretable to human observers.[145] Each action can be audited through examination of the articulated cognitive process preserved in the interaction history. In contrast, the reasoning of an RL agent remains implicit, encoded as subsymbolic activation patterns distributed across the numerical weights of its policy network. The mechanisms for guiding agent behavior also differ markedly: LLM agents respond to rich, semantic feedback expressed in natural language instructions and structured tool outputs, while RL agents are directed almost exclusively by scalar reward signals, a comparatively sparse form of supervision from which effective long-term strategies must be inductively inferred.

An agentic system can now be formalized as a discrete-time stateful process designed to accomplish specified objectives through continuous environmental interaction. The system's behavior is characterized by an iterative loop mediated by a core generative model. The complete system can be rigorously defined as a tuple:

$$\mathbf{S} = (\mathbf{H}, \mathbf{T}, E, \pi)$$

where each component is specified as follows:

- **Interaction History** $H_t \in \mathbf{H}$: This component represents the agent's episodic memory and constitutes its complete state representation at discrete time step t. It is formally defined as a chronologically ordered sequence of Cognition-Action-Observation tuples:

  $$H_t = [\langle C_1, A_1, O_1 \rangle, \ldots, \langle C_{t-1}, A_{t-1}, O_{t-1} \rangle]$$

  The space of all possible histories $\mathbf{H}$ is the Kleene closure of the tuple space:

  $$\mathbf{H} = (\mathbf{C} \times \mathbf{A} \times \mathbf{O})^*$$

  where $\mathbf{C}$ denotes the space of cognitive states (including textual reasoning traces and internal deliberations), $\mathbf{A}$ represents the action space (all possible executable actions), and $\mathbf{O}$ defines the observation space (all possible environmental percepts).

- **Toolset** $\mathbf{T}$: A finite collection of available tools or primitive capa-

bilities, formally expressed as:

$$\mathbf{T} = \{t_1, t_2, \ldots, t_k\}$$

This toolset defines the agent's effective action space $\mathbf{A}$, where an action $A \in \mathbf{A}$ typically corresponds to the invocation of a tool $t_j \in \mathbf{T}$ with specific parameters. Functionally, $\mathbf{T}$ establishes a critical boundary contract that translates the non-deterministic, semantic outputs of the core model M into discrete, deterministic operations executable by the environment.

- **Executor** E: The environment dynamics function, modeling the world's response to agent actions. It is formally defined as a (potentially stochastic) mapping:

$$E : \mathbf{A} \to \mathbf{O}$$

This function encapsulates all external state transitions and information feedback resulting from an action's execution, effectively defining the agent's perceptual interface to its operational environment.

- **Policy** π: The agent's decision-making function, or control policy. At each time step t, it maps the current history, task specification, and available toolset to a new cognition-action pair:

$$\pi : \mathbf{H} \times \text{Task} \times \mathbf{T} \to \mathbf{C} \times \mathbf{A}$$

The policy π is not implemented as a monolithic function but rather as a composite information processing pipeline centered on the core generative model M. This decomposition reveals the internal cognitive architecture:

$$\pi = \psi \circ M \circ \rho$$

where the constituent functions are defined as:

- ρ **(State Encoder / Contextualization)**: A serialization function that transforms the agent's structured state representation, comprising its history H_t, task specification, and toolset $\mathbf{T}$, into a linearized textual prompt suitable for processing by the language model M.

This transformation must preserve all semantically relevant information while adhering to the model's input constraints.

- M **(Core Generative Model)**: The central computational engine, typically instantiated as a large language model. It performs a probabilistic sequence-to-sequence transformation, generating an output string that encodes both the subsequent cognitive trace C_t and action specification A_t, conditioned on the contextualized input provided by ρ.

- ψ **(Action Decoder / Response Parser)**: A deserialization function that extracts structured, machine-interpretable information from the model's raw textual output, producing a formal tuple $\langle C_t, A_t \rangle \in \mathbf{C} \times \mathbf{A}$. This parsing step is essential for grounding the model's generative capabilities into executable environmental actions.

The temporal evolution of the agent can be formalized as a discrete-time dynamical system. The system state at time t is its history H_t. The state transition from H_t to H_{t+1} proceeds through the following operational sequence:

1. **Policy Invocation**: The agent computes its next cognitive state C_t and external action A_t by applying its policy to the current state:

$$\langle C_t, A_t \rangle = \pi(H_t, \text{Task}, \mathbf{T})$$

2. **Environmental Interaction**: The selected action A_t is executed in the environment, yielding an observation O_t:

$$O_t = E(A_t)$$

3. **State Update**: The agent's history is augmented by appending the new experience tuple, forming the state for the subsequent iteration:

$$H_{t+1} = H_t \oplus \langle C_t, A_t, O_t \rangle$$

where $\oplus$ denotes the sequence concatenation operator.

This iterative process continues until a terminal condition is satisfied, such as explicit task completion, reaching a maximum step limit, or encountering an unrecoverable error state. This formalism directly instantiates the classical **Perceive-Think-Act** cycle from cognitive architecture theory and control systems, with the history H_t serving as the perceptual basis and the policy π encapsulating both deliberative reasoning and action selection phases.[126, 146]

The preceding formulation, describing a tool-calling agent, can be systematically extended to model more sophisticated program-generating agents. [26] This extension primarily redefines the action space $\mathbf{A}$ and correspondingly adapts the executor E. In a program-generating paradigm, an action A_t is no longer a simple structured tuple representing a single function call with discrete parameters. Instead, it is a complete program expressed as source code, potentially containing complex control flow constructs, conditional logic, loops, and orchestration of multiple sequential or parallel tool invocations.[130] Consequently, the executor E evolves from a simple function dispatcher into a full-fledged code interpreter or sandboxed runtime environment capable of executing arbitrary programs. The toolset $\mathbf{T}$ is not eliminated but is instead provided to the executor as a preloaded library of callable functions, accessible from within the generated program code.

This architectural shift necessitates modifications to the state encoder ρ. Rather than merely serializing the current state and available tool names, ρ must now include comprehensive API documentation for each tool in $\mathbf{T}$ within the prompt context. This documentation enables the core model M to generate syntactically valid and functionally correct programs that properly interface with the available toolset. The expanded prompt must specify function signatures, parameter types, return value semantics, and usage examples to guide correct code generation.

The primary advantage of the program-generating framework lies in its capacity to express sophisticated metastrategies as executable code. In this paradigm, the agent's reasoning is not confined to a single invocation of the core model M per decision cycle. Instead, the generated program can orchestrate an arbitrary sequence of chained calls to M to implement complex problem-solving algorithms dynamically. This capability is realized by exposing the agent's own policy π, or a functionally equivalent

wrapper around M, as a callable function t_M within the sandboxed execution environment E. The generated program can therefore recursively invoke the reasoning model at intermediate steps of its execution, using it as a computational primitive for generating content, making decisions, or performing analysis. This effectively transcends the simple **Think-Act** cycle and enables the programmatic implementation of dynamic, multi-step reasoning strategies.

This paradigm enables the implementation of various advanced reasoning patterns through explicit code rather than implicit prompting:

- **Scratchpad Reasoning**:[131] A program can implement an iterative loop that repeatedly calls M to generate the next step in a reasoning chain, accumulating intermediate results in a local variable that serves as evolving context for subsequent calls.

- **Chain-of-Thought Decomposition**:[191] Complex problems can be programmatically divided into subproblems, with each subproblem solved through a separate model invocation, and solutions composed through explicit algorithmic logic.

- **Recursive Self-Improvement**:[113] A program can generate an initial solution through one model call, invoke M again in a critique mode to evaluate that solution, and use the critique to guide a refinement call, iterating until convergence.

- **Tree-of-Thoughts Exploration**:[198] Parallel exploration strategies can be coded by generating multiple candidate solutions, evaluating them concurrently (potentially in parallel), and programmatically deciding which paths to pursue, prune, or merge based on intermediate assessments.

In all these cases, the generated program embodies the agent's metastrategy, using the core model M as its fundamental reasoning primitive while maintaining explicit control over the problem-solving process through conventional programming constructs.

While program generation grants agents unprecedented flexibility and expressive power, it simultaneously exposes profound challenges rooted in their computational substrate. Current methodologies predominantly

construct agent logic within high-level, general-purpose programming languages such as Python, JavaScript, or TypeScript. Though expedient for rapid prototyping and leveraging existing library ecosystems, this architectural choice creates fundamental limitations across three critical dimensions:

Portability: Agent logic implemented in high-level languages becomes inextricably coupled to the extensive surface area of the host language runtime and operating system. Dependencies on platform-specific libraries, filesystem conventions, and runtime assumptions prevent verifiable migration across heterogeneous environments. An agent designed for a Linux server cannot be confidently deployed to an embedded system, mobile device, or browser environment without substantial reimplementation.

Verifiability: The semantic ambiguity inherent in high-level languages, arising from non-deterministic garbage collection, dynamic typing with implicit coercions, complex mutable state semantics, and undefined behavior specifications, renders formal verification intractable.[134] The computational model lacks the mathematical precision necessary for proving correctness properties or establishing security guarantees through rigorous analysis.

Security: Perhaps most critically, agents executing within standard language runtimes inherit the full ambient authority of their host process. By default, such an agent possesses unrestricted access to any resource available to the user account under which it executes, including the entire filesystem, network interfaces, and system APIs. This ambient authority model presents an unacceptable security risk for autonomous systems, as a single vulnerability in the agent's logic or a successful prompt injection attack can be leveraged to compromise the entire host system.[99, 157]

The alternative security model, capability-based security,[41] offers a principled solution. In this paradigm, the computational substrate is, by default, completely isolated from external resources. The sandbox possesses no intrinsic capabilities to interact with the outside world. Every permission, from network access to filesystem operations, must be explicitly granted as a discrete, unforgeable capability token by a trusted host environment.[120, 121] This design ensures that the agent operates under the principle of least privilege, possessing only the minimal set of capabilities necessary for its designated function, thereby fundamentally limiting

its potential for unintended or malicious behavior.[150]

To surmount these obstacles, this work advocates for the development of a novel foundational substrate specifically architected for agentic computation. Incremental improvements to application-level frameworks are insufficient, as the core issues of security, verifiability, and portability are deeply embedded in the underlying execution model. Therefore, this chapter undertakes a principled design process, guided by first principles from programming language theory and formal methods, to establish the formal requirements for such a substrate. The objective is to define a computational model that satisfies four essential criteria:

1. **Minimalism**: The substrate should be Turing-complete but possess only those features strictly necessary for universal computation, thereby minimizing the attack surface and reducing semantic complexity.

2. **Formal Specification**: All aspects of the substrate's syntax and operational semantics must be rigorously specified using mathematical formalisms, enabling formal reasoning about program behavior and supporting mechanized verification.

3. **Inherent Safety**: The substrate must provide memory safety and process isolation as intrinsic properties enforced by its design, not as probabilistic guarantees provided by external sandboxing mechanisms.

4. **Platform Agnosticism**: The substrate must function as a universal compilation target, abstracting away the idiosyncrasies of source languages, CPU architectures, and host operating systems.

Following the establishment of these theoretical requirements, this chapter demonstrates that a carefully delineated subset of the WebAssembly standard, particularly its textual representation (WAT), serves as a practical instantiation of this ideal computational substrate. The argument proceeds by showing that WebAssembly, when properly constrained and augmented with minimal well-defined extensions, satisfies all specified criteria while providing access to a mature, industry-tested ecosystem of runtimes, compilers, and development tools. Building upon this foundation,

the discussion introduces two novel architectural enhancements: a principled value-based error-handling protocol and a secure input-output bridge for controlled data exchange between the sandboxed agent and its host environment. The chapter culminates in the presentation of a comprehensive reference implementation and evaluation demonstrating the framework's practical viability for developing secure, portable, and verifiable autonomous agents.

4.1 Formal Substrate Specification

Before evaluating existing technologies as candidate substrates for agentic computation, it is necessary to establish a rigorous set of requirements derived from first principles. These requirements emerge from established principles in programming language theory, operational semantics, and abstract machine design. The objective is to characterize the simplest possible system that satisfies the core design goals of safety, verifiability, and portability, while remaining sufficiently expressive for Turing-complete computation. This formal specification will subsequently serve as a benchmark against which any practical implementation can be evaluated for suitability.

4.1.1 Design Requirements

The architecture of this computational substrate is governed by several fundamental design principles, each addressing a critical limitation of contemporary agent frameworks. These principles collectively define what it means for a substrate to be suitable for secure, verifiable, and portable agentic computation.

Principle 1: Inherent Safety. The substrate must provide memory safety and process isolation as foundational guarantees enforced by its operational semantics, not as probabilistic properties provided by external mechanisms. This requirement has several concrete implications. First, code executing within the substrate must be axiomatically incapable of accessing memory outside its explicitly allocated regions, thereby eliminating entire classes of vulnerabilities such as buffer overflows, use-after-free errors, and arbitrary memory corruption. Second, the substrate

must enforce strict isolation between distinct computational units, ensuring they can only interact through well-defined, mediated interfaces rather than through shared mutable state. Third, these safety properties must be verifiable through static analysis prior to execution, not merely detected through runtime checks that impose performance overhead and provide probabilistic rather than absolute guarantees.

Crucially, this safety must be intrinsic to the substrate's design rather than dependent on operating system-level sandboxing mechanisms. OS-level process isolation, while valuable, operates at too coarse a granularity and relies on complex kernel implementations whose correctness is difficult to verify. The substrate's safety guarantees must hold regardless of the host operating system, embedded environment, or execution context in which it is deployed. This requirement fundamentally distinguishes this approach from systems that achieve security through containerization or virtual machines, which add layers of complexity and potential failure modes.[100]

Principle 2: Well-Defined Semantics. Every aspect of the substrate's behavior must be precisely specified through formal semantics, eliminating ambiguity and undefined behavior. This principle is essential for creating a deterministic and verifiable platform where mathematical reasoning about program correctness is possible. The specification must define the substrate's behavior through rigorous operational semantics that describe how programs execute, what invariants are preserved, and what properties can be proven about execution traces.

This requirement stands in sharp contrast to languages like C or C++, where large swaths of behavior are explicitly designated as "undefined" by the language specification. Undefined behavior grants compiler implementers freedom to optimize aggressively but renders programs inherently non-portable and impossible to reason about formally. When a program invokes undefined behavior, literally anything can happen, from silent data corruption to security vulnerabilities, and behavior can vary arbitrarily across different compilers, optimization levels, or target architectures. For a substrate designed to support autonomous agents where correctness and security are paramount, such ambiguity is unacceptable.

The formal semantics must be complete, covering not only the "happy path" of correct execution but also precisely defining error conditions, resource exhaustion scenarios, and the behavior of all edge cases. This com-

pleteness enables formal verification: if every possible execution path is rigorously defined, it becomes possible to prove properties about program behavior using techniques from programming language theory, such as type safety theorems, termination analysis, and invariant preservation.[78]

Principle 3: Platform Independence. The substrate must function as a universal compilation target, abstracting away the particulars of source languages, underlying hardware architectures, and host operating systems. This requirement decomposes into several distinct facets. First, the substrate must not favor any particular high-level programming language or object model. Developers should be free to implement agent logic in C, Rust, Go, AssemblyScript, or any other language that can compile to the substrate's intermediate representation. The substrate serves as a common backend that enables interoperability between components written in different source languages.

Second, the substrate must be hardware-agnostic, with its semantics defined independently of specific CPU architectures such as x86, ARM, or RISC-V. While implementations will ultimately execute on physical hardware, the substrate's abstract machine model should not expose architecture-specific details like register allocation, cache hierarchies, or instruction set peculiarities. This abstraction ensures that agent logic compiled to the substrate is genuinely portable across the full spectrum of computing devices, from embedded microcontrollers to cloud servers.

Third, platform independence must extend to the embedding environment. The same compiled agent module should execute identically whether hosted in a web browser, a cloud function runtime, a desktop application, or a standalone virtual machine. This environmental agnosticism requires that the substrate make minimal assumptions about the host system, communicating with it only through a narrow, well-defined interface that can be implemented consistently across diverse platforms.

Principle 4: Efficient and Modular Representation. While the primary focus of this specification is on formal semantics rather than implementation efficiency, the abstract computational model must map to concrete representations that are compact for storage and transmission, and efficient to validate and execute. The design should enable one-pass validation, where a module can be verified for correctness in a single linear scan without requiring multiple passes or complex dependency resolution.

This property is essential for environments with constrained resources or strict latency requirements.

The substrate must support flexible compilation strategies. Some deployment scenarios may favor ahead-of-time (AOT) compilation to native machine code for maximum performance, while others may require just-in-time (JIT) compilation for dynamic loading, or pure interpretation for maximum portability and sandboxing guarantees. The substrate's design should not preclude any of these implementation strategies but should instead provide a sufficiently well-defined intermediate representation that supports multiple compilation backends.

Modularity is also essential to the design. Programs must be decomposable into smaller, independently verifiable units that can be developed, tested, and validated in isolation. This modularity supports code reuse, enables incremental compilation and caching strategies, and provides the foundation for dynamic linking and runtime capability management. A module system also establishes the natural boundary at which security policies are enforced, with each module explicitly declaring its dependencies and the capabilities it requires from the host environment.

Textual Representation Requirements. While the executor E operates on the substrate's formal semantics, the programs it executes must be represented in a format that serves as the primary interface for the generative model M. This textual representation must satisfy two complementary objectives. First, its syntax must be simple, regular, and compositional, making it reliably producible by an LLM without requiring specialized training or fine-tuning. The representation should leverage syntactic patterns and structural regularities that LLMs naturally learn during pretraining on diverse code corpora, such as S-expressions, hierarchical indentation, or explicit delimiters for nested structures.

Second, despite being machine-targeted, the format must remain human-readable and transparent. This readability is not a mere convenience but a security requirement: it must be possible for human developers to directly inspect, analyze, and audit the agent's generated logic without requiring decompilation or sophisticated tooling. The textual format serves as the primary artifact for security reviews, debugging sessions, and formal verification workflows. Opacity in this representation would undermine the entire framework's goal of building trustworthy, auditable agents.

These dual requirements, machine-friendliness and human-readability, suggest a textual format that is more structured than typical high-level source code but less opaque than binary machine code. The representation should strike a balance: sufficiently low-level to have unambiguous semantics and efficient execution, yet sufficiently high-level to be comprehensible and analyzable by human experts. This balance is critical for enabling the collaborative human-AI development model where AI generates programs and humans verify their correctness and safety.

4.1.2 Abstract Computational Model

In accordance with foundational principles of programming language theory, the substrate is defined through its abstract syntax rather than a particular concrete textual or binary encoding. Abstract syntax provides a formal description of the structural components and composition rules of programs, independent of superficial syntactic details like whitespace, keyword choices, or delimiter styles. This separation of essential structure from incidental surface syntax is fundamental to principled language design. It enables rigorous reasoning about program semantics, supports multiple concrete syntaxes that map to the same abstract structure, and facilitates formal proofs about language properties.

Type System: The Foundation of Safety. A defining characteristic of this substrate's minimalist design is its radically simplified type system. Most programming languages feature elaborate type hierarchies with primitives for integers of various widths, floating-point numbers, characters, booleans, and complex aggregate types like structures and unions. This substrate, in contrast, provides exactly one fundamental value type: the 32-bit unsigned integer, denoted `u32`. This extreme parsimony is not a limitation but a deliberate design choice that follows from the goal of minimal yet universal computation.

Within low-level computational systems, a fixed-width integer serves as a universal building block from which all other data types can be constructed through interpretation. By restricting all computation to operations on `u32` values, several critical objectives are achieved. First, the instruction set becomes dramatically simplified, as every operation has a uniform operand type. Second, the validation rules of the abstract machine become tractable, with no complex subtyping relationships, generic instan-

tiations, or type inference algorithms to implement. Third, the reduced surface area minimizes opportunities for type confusion vulnerabilities or implementation bugs in type checking logic.

The semantic interpretation of a `u32` value is not intrinsic to its bit representation but is imposed by the instructions that operate upon it. The same 32-bit pattern can represent:

- An unsigned integer $n \in [0, 2^{32} - 1]$ for arithmetic operations
- A signed integer in two's complement representation for operations requiring sign extension
- A byte offset into linear memory for load and store instructions
- An error code or status value returned from fallible operations
- An opaque handle or capability token representing a host-managed resource
- A boolean value, where 0 represents false and non-zero represents true
- An encoded enumeration or discriminated union tag

This interpretive flexibility, where the same bit pattern acquires different meanings in different computational contexts, is a hallmark of low-level systems and enables the construction of complex data structures and abstractions entirely within the `u32` universe.

Computational Model: Stack Machine Architecture. The substrate adopts a stack machine computational model, a choice motivated by simplicity, verifiability, and efficient implementation. In a stack machine, operands for instructions are implicitly drawn from an operand stack rather than named registers, and results are pushed back onto this stack. This architecture eliminates the complexity of register allocation, a notoriously difficult problem in compiler construction, by avoiding named storage locations for intermediate values entirely.

The stack discipline provides strong structural guarantees that facilitate static analysis. At any point in program execution, the stack depth and the types of values on the stack are statically determinable from the

program text, without requiring symbolic execution or constraint solving. This property enables one-pass validation and trivializes certain classes of errors: stack underflow, type mismatches, and many control-flow errors can be detected through purely syntactic analysis.

The instruction set architecture decomposes into functionally coherent categories:

Numeric Instructions provide comprehensive arithmetic and logical operations on `u32` values. This category includes:

- Constants: `const` pushes a literal value onto the stack
- Comparisons: `eq`, `ne` (equality), `lt_s`, `lt_u`, `gt_s`, `gt_u`, `le_s`, `le_u`, `ge_s`, `ge_u` (signed and unsigned relational tests)
- Arithmetic: `add`, `sub`, `mul`, `div_s`, `div_u`, `rem_s`, `rem_u` (basic operations with signed and unsigned variants)
- Bitwise: `and`, `or`, `xor`, `shl`, `shr_s`, `shr_u`, `rotl`, `rotr` (logical and shift operations)
- Bit manipulation: `clz`, `ctz`, `popcnt` (count leading zeros, trailing zeros, and set bits)

Memory Instructions mediate interaction with the sandboxed linear memory space:

- `load`: Reads a value from a specified memory address
- `store`: Writes a value to a specified memory address
- `memory.size`: Returns the current size of linear memory in units of 64KB pages
- `memory.grow`: Requests expansion of linear memory by a specified number of pages

Variable Instructions provide access to named storage locations that persist across instruction boundaries:

- `local.get`, `local.set`, `local.tee`: Access function-local variables

- `global.get`, `global.set`: Access module-global variables

The critical design decision concerns control flow. The substrate employs *structured control flow* exclusively, a constraint that is essential for verifiability. Arbitrary jumps, such as `goto` instructions that can transfer control to any labeled location in the program, are explicitly prohibited.[45] Instead, execution flow is managed through well-nested, lexically scoped constructs:

- `block`, `end`: Define a labeled block scope
- `loop`, `end`: Define a labeled loop scope where branches target the loop head
- `if`, `else`, `end`: Provide conditional execution with optional else branch

Branching instructions can only target these enclosing structured constructs:

- `br`: Unconditional branch to a labeled construct
- `br_if`: Conditional branch based on stack top
- `br_table`: Indexed branch to one of several labels (switch statement)

This structured control flow constraint ensures that the program's control-flow graph is *reducible*, meaning it contains no irreducible loops or pathological control structures. Reducibility is a crucial property for static analysis, optimization, and verification. It guarantees that the program can be analyzed using standard dataflow analysis techniques, that dominance relationships are well-defined, and that loop structures are identifiable without complex graph algorithms.

Function invocation is handled through two mechanisms:

- `call`: Direct invocation of a statically-known function by index
- `call_indirect`: Dynamic dispatch through a function pointer table, enabling higher-order programming patterns

The complete instruction set, detailed in Table 4.1, provides a Turing-complete set of operations while maintaining minimal surface area. This minimalism is not restrictive: any computable function can be expressed using these primitives, but the reduced instruction count dramatically simplifies the formal specification and verification of the substrate's semantics.

Table 4.1. Substrate Instruction Set

Instruction(s)	Description
Numeric Instructions	
const	Push 32-bit constant to stack.
eqz, eq, ne	Equality/inequality tests.
lt_s/u, gt_s/u, le_s/u, ge_s/u	Signed/unsigned comparisons.
clz, ctz, popcnt	Count leading/trailing zeros or set bits.
add, sub	Integer addition, subtraction.
mul	Integer multiplication.
div_s/u, rem_s/u	Signed/unsigned division, remainder.
and, or, xor	Bitwise logical operations.
shl, shr_s/u, rotl, rotr	Bitwise shift/rotate operations.
Memory Instructions	
load	Load value from memory.
store	Store value to memory.
memory.{size, grow}	Query/expand linear memory.
Variable and Table Instructions	
local.{get, set, tee}	Read/write local variables.
global.{get, set}	Read/write global variables.
table.{get, set}	Access indirect call table.
Control Instructions	
unreachable	Forces an unconditional program trap.
nop	Performs no operation.
block, loop	Define structured instruction blocks.
if, else	Provide conditional execution paths.
end	Marks the end of a structured block.
br, br_if, br_table	Direct, conditional, and indexed branching.
return	Return from the current function.
call, call_indirect	Direct or indirect function call.
drop	Discards the top value on the stack.
select	Selects one of two values based on a condition.

State Management: Linear Memory Model. State persistence across instruction execution is managed through interaction with a single, sandboxed linear memory space. This memory is modeled as a contiguous, byte-addressable array that is isolated from the host system and from other module instances executing concurrently. The linear memory model provides several critical properties. First, it is *flat*, with no segmentation, paging, or virtual address translation visible to the program. Every byte has a unique, zero-based offset in $[0, N-1]$ where N is the current memory size.

Second, it is *bounded*. Since memory addresses are represented by `u32` values, the maximum addressable memory space is 2^{32} bytes, or 4 GiB. This finite upper bound is not merely an implementation convenience but a fundamental property that enables static reasoning about resource usage and prevents entire classes of denial-of-service attacks based on unbounded memory allocation. Memory is allocated in units of *pages*, where each page is exactly 65,536 bytes (64 KiB). This page granularity matches common OS page sizes and provides a convenient unit for memory management.

Third, it is *isolated*. The linear memory of one module instance cannot be accessed by another module or by the host except through explicitly mediated operations. This isolation is enforced by the substrate's operational semantics: there are no instructions or mechanisms for a module to reference memory outside its own linear address space. All interactions with external state must occur through imported host functions that serve as controlled, auditable gateways.

The memory instruction family provides the interface to this linear memory:

- `load` variants read 1, 2, or 4 bytes from a specified address and push the value onto the operand stack
- `store` variants pop a value from the stack and write 1, 2, or 4 bytes to a specified address
- `memory.size` returns the current memory size in page units
- `memory.grow` attempts to expand memory by a specified number of pages, returning the previous size on success or -1 on failure

All memory accesses are bounds-checked at runtime. Any attempt to access an address outside the currently allocated memory region results in an immediate, controlled program termination (trap) rather than undefined behavior or silent data corruption. This bounds checking is the primary mechanism enforcing memory safety in the substrate.

Modularity and Interface Boundaries. The fundamental unit of code organization, deployment, and security enforcement is the *module*. A module is a self-contained entity that encapsulates a collection of definitions, including:

- **Functions**: The executable code implementing the module's behavior
- **Linear Memory**: The module's private data storage
- **Global Variables**: Module-scoped state visible across all functions
- **Tables**: Indirect function call tables for dynamic dispatch

Critically, a module declares its interface with the external world through two explicit lists:

- **Imports**: Specifies the functions, memories, globals, and tables that the module requires the host environment to provide. Each import is identified by a two-level namespace (module name, item name) and includes a type signature that the host must satisfy. Imports define the module's *dependencies* and represent the sole channel through which sandboxed code can access external capabilities.
- **Exports**: Declares the definitions that the module makes available to the host. Exports define the module's *public API* and enable composition, where one module's exports can satisfy another module's imports.

This explicit import/export mechanism establishes a clear, formally specified boundary between the module's internal implementation and its external interface. This boundary is the cornerstone of the substrate's capability-based security model. A module cannot access any external resource unless that resource is explicitly provided to it as an import. There is no ambient authority, no global namespace of functions that modules

can invoke without declaration, and no mechanism for a module to "discover" or access capabilities that were not explicitly granted to it.

The module system supports *compositional reasoning*: each module can be analyzed, validated, and verified independently, with its dependencies represented abstractly by their type signatures. This modularity is essential for scalability, enabling large agent systems to be constructed from smaller, auditable components, and for dynamic loading, where new modules can be introduced at runtime after independent verification.

4.1.3 Execution Semantics

Having defined the static structure of programs through abstract syntax, their dynamic behavior is now formalized through operational semantics. The framework of *small-step operational semantics*[137] is adopted, a standard approach from programming language theory that models program execution as a sequence of discrete atomic state transitions within an abstract machine. This formalism provides a rigorous, mathematical foundation for reasoning about how programs behave at runtime, enabling formal proofs of properties like type safety, determinism, and resource bounds.

Machine State: The Configuration. The complete instantaneous state of a computation is captured by a *configuration*, denoted as a triple $\langle S, F^*, \text{instr}^* \rangle$, where:

The **store** S serves as the global repository for all allocated runtime objects. It is a mapping from addresses to object instances:

$$S : \text{Address} \to \{\text{FuncInst}, \text{MemInst}, \text{GlobalInst}, \text{TableInst}\}$$

Each instance type represents a runtime instantiation of the corresponding module definition:

- FuncInst: A function instance, containing the function's code and a reference to its parent module instance for resolving imports
- MemInst: A memory instance, containing the byte array and current size
- GlobalInst: A global variable instance, containing a mutable or immutable `u32` value

- TableInst: A table instance, containing an array of function references for indirect calls

The store is conceptually similar to a heap in conventional programming languages but is more structured. Objects in the store are never deallocated during module execution, eliminating concerns about use-after-free or dangling pointers. The store grows monotonically as new modules are instantiated or existing modules allocate additional resources.

The **frame stack** F^* represents the call stack, where each frame F corresponds to an active function invocation. A frame contains:

$$F = \langle \text{locals}, \text{module\_inst} \rangle$$

where locals is an array of `u32` values representing the function's local variables (including parameters), and module_inst is a reference to the module instance from which the function originated. This module reference is essential for resolving references to other definitions (functions, globals, memories) within the same module's namespace.

The **instruction sequence** instr* represents the remaining instructions to be executed. This sequence is not merely the static program text but includes the implicit operand stack, represented as a prefix of constant instructions. For instance, the abstract instruction sequence ((`const` 5)(`const` 3)`add`) represents the concrete state where values 5 and 3 are on the operand stack, with an `add` instruction ready to execute.

Reduction Rules: Atomic State Transitions. Execution proceeds by repeatedly applying *reduction rules* to transform configurations. A reduction rule has the general form:

$$S; F; \text{instr}^* \to S'; F'; \text{instr'}^*$$

which specifies that the configuration $\langle S, F, \text{instr}^* \rangle$ transitions in a single atomic step to the configuration $\langle S', F', \text{instr'}^* \rangle$. The complete semantics is defined by providing reduction rules for every instruction.

For simple arithmetic, the store and frame remain unchanged, and the rule describes pure manipulation of the operand stack. For example, addition is formalized as:

$$S; F; (\texttt{const}\ n_1)(\texttt{const}\ n_2)\texttt{add}\ \text{instr}^* \to S; F; (\texttt{const}\ n)\ \text{instr}^*$$

where $n = (n_1 + n_2) \bmod 2^{32}$. This rule states that when two constant values appear before an `add` instruction, they are replaced by their sum, computed modulo 2^{32} to handle overflow.

Memory operations modify the store. For instance, storing a value has the form:

$$S; F; (\texttt{const}\ \text{addr})(\texttt{const}\ v)\texttt{store}\ \text{instr}^* \to S'; F; \text{instr}^*$$

where S' is identical to S except that byte at address addr in the current frame's memory instance is updated to value v. If addr is out of bounds, the reduction instead transitions to a trap state, which represents program termination.

Control flow instructions manipulate the instruction sequence and frame stack. A function call creates a new activation frame:

$$S; F; (\texttt{const}\ a_1) \cdots (\texttt{const}\ a_n)(\texttt{call}\ f)\ \text{instr}^* \to S; F' \cdot F; \text{body}_f$$

where F' is a new frame with locals initialized to the argument values $a_1, \ldots, a_n$, and body_f is the instruction sequence of the called function. The original frame F is pushed onto the frame stack, awaiting the callee's return.

A return instruction reverses this process:

$$S; F' \cdot F; (\texttt{const}\ r_1) \cdots (\texttt{const}\ r_m)\texttt{return}\ \text{instr}^* \to S; F; (\texttt{const}\ r_1) \cdots (\texttt{const}\ r_m)\ \text{instr}^*$$

The top frame F' is discarded, returning control to the caller frame F, with the return values pushed onto the caller's operand stack.

Branching instructions like `br` unwind the instruction sequence to a labeled block:

$$S; F; (\texttt{block}\ \tau\ \text{instr}_b^*)\ \text{instr}_a^* \to S; F; \text{instr}_a^*$$

when a `br` targeting this block is executed. The precise semantics depend on the block structure and nesting depth of the branch target, but the key property is that control flow is always structured: branches can only exit to lexically enclosing blocks.

Validation and Type Safety. A critical aspect of this semantic model is the distinction between *validation* (static semantics) and *execution* (dynamic semantics). Before any module can be executed, it must undergo a validation phase. Validation is a static analysis process that type-checks the module's code without executing it, verifying several critical properties:

1. **Stack Discipline**: Every instruction consumes and produces values on the operand stack in a manner consistent with its type signature. The stack is never underflowed (popping from an empty stack) or overflowed (unbounded growth).

2. **Type Consistency**: Instructions receive operands of the expected types. For instance, `add` requires two `u32` values on the stack.

3. **Control Flow Validity**: All branch targets are valid, referencing lexically enclosing blocks. Function calls reference valid function indices with matching type signatures.

4. **Memory Safety**: Memory accesses are properly aligned and memory indices are within declared bounds (though specific addresses are validated at runtime).

5. **Resource Bounds**: The module declares its resource requirements (initial/maximum memory size, table sizes) and does not exceed them.

If a module passes validation, the type system provides a powerful *soundness guarantee*: the program will never enter an undefined state during execution. More formally, this is expressed through two properties:

- **Progress**: A well-typed program either reaches a final value configuration or can take another reduction step. It never gets "stuck" in a state where no rule applies.

- **Preservation**: If a configuration is well-typed and reduces to another configuration, that new configuration is also well-typed with consistent types.

These guarantees are the formalization of memory safety and type safety. They ensure that validated programs cannot corrupt memory, cannot forge pointers, and cannot violate the module boundary to access unauthorized resources. This upfront validation is what enables runtimes to perform aggressive optimizations: many runtime checks can be safely eliminated because the validator has proven they are unnecessary.

Host Interaction and Store Extension. The final component of the operational semantics concerns interaction with the external world through imported host functions. Since host functions are implemented outside the substrate in the host language, their behavior cannot be defined by the reduction rules. Instead, their invocation is modeled as a non-deterministic transition:

$$S; F; \text{args}^* \ (\texttt{call}\ f_{\text{host}})\ \text{instr}^* \rightarrow_{\text{host}} S'; F; \text{results}^*\ \text{instr}^*$$

This transition represents the host function executing with the provided arguments and returning results. The subscript $_{\text{host}}$ indicates that this reduction is governed by the host's specification rather than the substrate's rules. To maintain the soundness of the overall system, a critical constraint is imposed on how the store can change:

$$S \preceq S'$$

This *store extension* property specifies that the host function can allocate new objects (functions, memories, globals) or modify existing mutable objects (memory contents, mutable globals), but it cannot:

- Delete or invalidate existing objects
- Modify immutable objects (functions, immutable globals)
- Violate type consistency of existing objects
- Access or corrupt the internal state of other module instances

This constraint ensures that host functions, regardless of their internal implementation complexity, cannot undermine the invariants that the abstract machine maintains about its own state. The host may introduce new capabilities or mutate shared state in controlled ways, but it cannot

violate the module's sandbox integrity. This formalization of host interaction is crucial for reasoning about composed systems where untrusted agent modules interact with trusted host capabilities.

The store extension property also provides a clean separation between the *pure* substrate semantics and the *impure* effects performed by the host. The substrate's reduction rules are deterministic and effect-free, operating only on abstract values and memory within the sandbox. All observable effects, network communication, filesystem access, user interface updates, are encapsulated entirely within host function implementations. This separation is not merely an implementation detail but a fundamental architectural principle that enables formal verification of the pure computational core while delegating effectful operations to independently audited host components.

Execution Model: From Modules to Running Programs. The complete execution lifecycle of a program proceeds through several distinct phases:

1. **Module Loading**: The module is parsed from its textual or binary representation into an abstract syntax tree.

2. **Validation**: The module undergoes static validation to verify type correctness, stack discipline, control flow validity, and resource bounds. Invalid modules are rejected before instantiation.

3. **Instantiation**: The validated module is instantiated by:

   - Allocating space in the store for its functions, memory, globals, and tables
   - Resolving its imports by linking them to host-provided capabilities
   - Initializing memory and tables with any specified constant data
   - Executing the module's start function, if present

4. **Execution**: The host invokes an exported function from the module, providing arguments and receiving results. This invocation creates an initial configuration with an empty frame stack, the arguments on the operand stack, and the function's body as the instruction

sequence. The abstract machine then repeatedly applies reduction rules until reaching a final state (successful return or trap).

5. **Termination**: Execution concludes either through normal return (the function completes and returns values) or through trap (an error condition is encountered). On normal return, results are passed back to the host. On trap, control returns to the host with an error indication, and the module instance may be discarded.

This lifecycle model, combined with the formal operational semantics, provides a complete specification of how programs execute. Every observable behavior of a program can be derived from this formal model, enabling rigorous reasoning about correctness, security properties, and resource usage. The formalism also serves as an unambiguous specification for runtime implementers, ensuring that different implementations of the substrate will execute the same program with identical observable behavior, modulo host function implementations.

Semantic Properties and Design Implications. Several crucial properties emerge from this semantic framework. First, *determinism*: given a fixed initial configuration and fixed host function implementations, the execution trace is completely determined. There is no source of non-determinism within the substrate itself, though host functions may introduce non-determinism through interaction with external systems. This determinism is essential for reproducibility, debugging, and formal verification.

Second, *isolation*: a module cannot observe or affect the state of other modules except through explicitly shared imports/exports mediated by the host. This isolation prevents interference between concurrently executing modules and enables strong sandboxing guarantees. Even if a module is malicious or buggy, its effects are contained within its own sandbox.

Third, *compositionality*: modules can be reasoned about independently. The behavior of a module depends only on its own code and the specifications of its imported functions, not on the implementations of other modules. This enables modular verification where each component is proven correct in isolation, and these proofs compose to establish system-wide properties.

Fourth, *resource accountability*: all resources consumed by a module

(memory, table space, call stack depth) are explicitly tracked and bounded. The host can enforce quotas and detect resource exhaustion before it impacts system stability. This accountability is essential for multi-tenant environments where untrusted modules share computational resources.

These properties collectively establish the substrate as a principled foundation for secure, verifiable computation. The formal semantics provide not merely a description of execution but a mathematical object that can be analyzed, proven correct, and used as a specification for implementation. This level of rigor distinguishes the substrate from ad-hoc runtime environments and enables the strong security and correctness guarantees required for autonomous agent systems.

4.2 WebAssembly Implementation

The abstract formalism developed in the preceding section establishes the theoretical requirements for an ideal computational substrate. The practical realization of this specification is now addressed, demonstrating that a carefully circumscribed subset of the WebAssembly standard, expressed in its textual format (WebAssembly Text, or WAT), constitutes a suitable instantiation. This selection represents a deliberate strategic decision to ground the theoretical framework in a robust, industry-vetted, and actively maintained standard rather than developing a novel system from first principles. The core design of WebAssembly aligns directly with the established principles: its sandboxed execution model provides inherent safety guarantees, its behavior is founded upon rigorous formal specification ensuring well-defined semantics, and it is architected as a portable compilation target guaranteeing platform independence.

WebAssembly's origins lie in the browser environment, where it emerged as a high-performance, low-level compilation target designed to enable near-native execution of computationally intensive applications within web pages. This genesis in the security-critical browser context mandated stringent safety and portability properties from its inception. However, despite its web-centric origins, WebAssembly's fundamental design represents that of a general-purpose virtual instruction set architecture, not a technology intrinsically bound to any particular deployment environment. Its utility has expanded substantially beyond the browser into diverse contexts

including server-side applications, edge computing platforms, embedded systems, and cloud-native infrastructure, largely catalyzed by initiatives such as the WebAssembly System Interface (WASI). This broad applicability across the computing spectrum makes it an ideal foundation for a universal agentic framework. By building upon WebAssembly, this work leverages a technology that is rapidly becoming a standard abstraction layer in the modern computing stack, ensuring that resulting systems benefit from portability, durability, and access to a mature ecosystem.[200]

It is essential, however, to distinguish the approach adopted in this work from the more generalized model embodied by WASI. The WebAssembly System Interface is designed to provide a comprehensive, POSIX-like compatibility layer that enables the portability of existing applications by offering an extensive API abstracting common operating system features. This includes filesystem access with hierarchical directory structures, network socket primitives, environment variables, command-line arguments, random number generation, and clock access. By providing this broad interface, WASI aims to serve as a universal system interface enabling unmodified Unix-style applications to execute portably across diverse host environments.

This breadth, while valuable for application portability, entails a correspondingly large and predefined API surface area. The formalism presented in this work, by contrast, mandates a minimal trusted computing base where the host environment exposes only a bespoke, application-specific set of capabilities precisely tailored to the agent's requirements. Rather than adopting a standardized but extensive interface like WASI, this framework defines its host interaction model through a minimal, explicitly curated manifest of functions. Each capability is deliberately granted, documented, and auditable. This ensures that the contract between guest and host adheres to the principle of least privilege: the agent possesses only those capabilities strictly necessary for its designated function, and each capability can be independently reviewed and verified. This architectural difference reflects a fundamental distinction in design philosophy: WASI prioritizes compatibility and convenience through comprehensive interfaces, while this framework prioritizes security and verifiability through minimal, explicit capability grants.

The mapping from the abstract formalism to a concrete WebAssembly

subset is systematic and direct. The single value type, `u32`, corresponds precisely to WebAssembly's `i32` type, a 32-bit integer. In strict adherence to the minimalism principle, other standard WebAssembly types from this subset. This exclusion encompasses `i64` (64-bit integers), `f32` and `f64` (32-bit and 64-bit floating-point numbers), and the more recently introduced reference types and vector types. The WebAssembly specification guarantees that a program restricted exclusively to `i32` operations will execute correctly and identically on any compliant runtime, regardless of the host's native word size or floating-point capabilities.

The instructions enumerated in the abstract model map directly to their WebAssembly counterparts with mechanical precision. The abstract `add`, `load` instructions correspond to WebAssembly's `i32.add`, `i32.load`, and the structured control constructs (`block`, `loop`, `if`) to the identically named WebAssembly instructions. The WebAssembly module system, with its explicit import and export sections defining interface boundaries, provides a direct implementation of the capability-based security model required by the formalism. A module's imports declare exactly which external functions it requires, and the host is responsible for satisfying these imports by providing implementations. This explicit declaration mechanism ensures that capabilities are never ambient but always explicitly granted through the module linkage process.

Crucially, the formal operational semantics of WebAssembly, as rigorously specified in its official specification document, satisfy the requirements for well-defined execution. The WebAssembly specification is formalized using small-step operational semantics, the same framework employed in the abstract model. This formalization is not merely documentation but has been mechanized and validated using the Isabelle/HOL theorem prover, providing machine-checked proofs of key safety properties including type soundness and memory safety. By adopting a WebAssembly subset, this does not introduce a novel computational model but rather leveraging a thoroughly studied, formally verified, and standards-compliant portion of an existing specification.

This approach of adopting a subset rather than the entirety of a standard provides significant strategic advantages. First, it grants immediate access to the extensive WebAssembly ecosystem. This ecosystem includes highly optimized production-grade runtimes such as Wasmtime (main-

tained by the Bytecode Alliance), Wasmer, and WasmEdge, each offering different performance characteristics and platform support. It encompasses sophisticated compiler toolchains including LLVM's WebAssembly backend (enabling compilation from C, C++, Rust, and other languages), AssemblyScript (a TypeScript-to-WebAssembly compiler), and Emscripten (a complete toolchain for porting existing C/C++ codebases). The ecosystem also provides extensive analysis tools for validation, optimization, debugging, and security auditing.

Second, this approach establishes a clear evolutionary path for future capability expansion. If application requirements evolve to necessitate features beyond the initial subset, such as 64-bit addressing for very large data structures or floating-point arithmetic for scientific computing, the subset can be judiciously expanded to incorporate standard WebAssembly extensions like `i64` or `f64` types. These extensions are already part of the formally specified standard, ensuring that their semantics are well-defined and their implementations are mature and tested. This incremental expansion capability allows the framework to adapt to emerging requirements without abandoning its foundational principles or requiring wholesale redesign.

Third, the subset approach provides a mechanism for controlled adoption of post-MVP (Minimum Viable Product) WebAssembly proposals as they mature and achieve standardization. The WebAssembly specification evolves through a rigorous standardization process managed by the W3C WebAssembly Working Group, where new features are proposed, formally specified, implemented in multiple engines, and thoroughly tested before achieving standard status. By explicitly identifying which extensions are adopted, clarity is maintained about the dependency surface while benefiting from the standardization community's collective wisdom and engineering effort.

However, the base WebAssembly MVP specification is not adopted without modification or augmentation. For the specific demands of a high-performance agentic framework, this work mandates the inclusion of several post-MVP extensions that have achieved widespread implementation and de facto standard status:

Multi-Value Returns: The MVP WebAssembly specification restricts functions to returning at most one value. The multi-value exten-

sion lifts this restriction, enabling functions to return tuples of values of arbitrary arity. This extension is essential for the error-handling protocol, detailed in the subsequent section, which relies on functions returning both a result value and an error code as a pair. Without multi-value support, this pattern would require awkward workarounds such as packing multiple values into memory and returning a pointer, which would compromise both performance and code clarity.

Bulk Memory Operations: The bulk memory proposal introduces instructions for efficient initialization and copying of large contiguous memory regions. Key instructions include `memory.copy` (copying between memory regions), `memory.fill` (setting a region to a constant byte value), and `memory.init` (initializing memory from a data segment). These operations are implemented as intrinsics in WebAssembly runtimes, often leveraging platform-specific optimizations like SIMD instructions or specialized CPU instructions (`memcpy`, `memset`). For data-intensive agentic tasks involving large context windows, document processing, or batch operations, these bulk operations provide substantial performance improvements over byte-by-byte loops.

Multiple Memories: The MVP specification restricts each module to at most one linear memory instance. The multiple memories extension lifts this limitation, allowing a module to declare and operate upon multiple distinct, isolated memory spaces. This capability is critical for the IO Bridge architecture detailed in Section 4.3.2. By providing separate memories for application state and host-guest communication, a stronger security boundary: the agent's working memory is physically separated from the data ingress/egress channels, preventing entire classes of memory corruption vulnerabilities. Each memory is accessed through explicitly indexed instructions (`memory.load` with a memory index), ensuring that confusion between memory spaces is detectable through static analysis.

These three extensions represent a minimal augmentation to the MVP specification, each addressing a specific architectural requirement while maintaining the formal rigor and safety properties of the base standard. All three have achieved Phase 4 (Standardize) status in the WebAssembly standardization process, indicating mature specification, multiple independent implementations, and comprehensive test suites. Their adoption does not compromise the principle of minimalism but rather recognizes that

certain higher-level constructs, when properly specified and implemented, provide substantial practical benefits while remaining within a formally verifiable framework.

The rationale for excluding other WebAssembly features deserves explicit articulation. Floating-point types are omitted types (`f32`, `f64`) because agentic control logic rarely requires non-integral arithmetic, and their inclusion would substantially complicate the validation semantics with IEEE 754 floating-point edge cases (NaN handling, rounding modes, denormals). 64-bit integers are excluded integers (`i64`) to maintain strict 2^{32} bounds on addressable memory and countable resources, simplifying resource accounting. Reference types are omitted types and garbage collection proposals because agent memory management is explicit and deterministic, avoiding the non-determinism introduced by automatic memory management. SIMD operations are excluded (vector) operations because, while valuable for data-parallel numerical computation, they are not essential for the symbolic and control-oriented processing that characterizes agentic behavior.

The textual representation of this subset is WebAssembly Text format (WAT), a human-readable, S-expression-based syntax for WebAssembly modules. WAT provides several properties essential for this framework. First, it is *unambiguous*: each WAT program has a unique, well-defined semantics specified by the WebAssembly standard. Second, it is *compositional*: complex programs are built from simpler nested constructs following regular syntactic patterns, making it amenable to generation by LLMs that have learned structural composition during pretraining. Third, it is *readable*: despite being a low-level representation, WAT preserves sufficient structure and uses mnemonic instruction names that enable human comprehension and auditing without requiring sophisticated decompilation tools.

An illustrative WAT fragment demonstrating the subset's expressiveness appears below:

```
(module
  (import "host" "read_sensor" (func $read_sensor (result i32)))
  (import "host" "actuate" (func $actuate (param i32)))
  (memory 1)
  (func $control_loop (export "run")
```

```
    (local $sensor_value i32)
    (local $threshold i32)

    ;; Read sensor value
    (call $read_sensor)
    (local.set $sensor_value)

    ;; Load threshold from memory
    (i32.load (i32.const 0))
    (local.set $threshold)

    ;; Compare and conditionally actuate
    (if (i32.gt_u (local.get $sensor_value)
                  (local.get $threshold))
      (then
        (call $actuate (i32.const 1))
      )
    )
  )
)
```

This example demonstrates several key properties: explicit imports declaring dependencies on host functions (`read_sensor`, `actuate`), local variable declarations with the restricted `i32` type, structured control flow using `if`/`then` blocks, and an exported function (`run`) serving as the module's entry point. The program's behavior is entirely deterministic and its resource requirements (one memory page) are statically declared.

The WebAssembly binary format, while not the primary interface for LLM generation, provides an important complementary representation. WAT programs are compiled to a compact binary encoding (.wasm files) for efficient storage, transmission, and execution. This binary format uses variable-length integer encodings and carefully designed opcodes to minimize size, typically achieving 20-30

The ecosystem tooling surrounding WebAssembly provides substantial infrastructure for this framework. The `wat2wasm` tool (part of the WebAssembly Binary Toolkit, WABT) compiles WAT to binary and validates the result. The `wasm2wat` tool performs the inverse transforma-

tion for debugging and inspection. The `wasm − objdump` tool provides detailed inspection of binary structure, including section contents, import/-export tables, and disassembled code. Runtime engines like Wasmtime offer command-line interfaces for instantiating and executing modules, detailed instrumentation for profiling and debugging, and programmatic APIs for embedding WebAssembly execution within host applications.

This mature toolchain integration stands in stark contrast to the position that would be occupied had a novel, bespoke substrate been developed. By anchoring on WebAssembly, decades of compiler optimization research (via LLVM), years of runtime engineering refinement (via production browser engines and standalone runtimes), and ongoing standardization effort ensuring long-term viability and evolution are inherited. This pragmatic reuse of existing infrastructure enables focus of engineering effort on the novel aspects of agentic computation—the capability model, error handling, and agent-specific tooling—rather than reimplementing fundamental compiler and runtime technology.

The formal verification story of WebAssembly provides additional assurance. The WebAssembly specification has been mechanized in the Isabelle/HOL theorem prover, yielding machine-checked proofs of type soundness (well-typed programs don't go wrong), progress (non-trapped programs can always take a step), and memory safety (programs cannot access memory outside their linear address space).[189] These proofs cover the entire MVP specification and have been extended to cover several post-MVP proposals. By building upon this formally verified foundation, this framework inherits these guarantees: any vulnerability in this system must arise from the additional architectural layers (the host functions provided, the error-handling convention, the IO Bridge) or from bugs in specific runtime implementations, not from ambiguities or flaws in the underlying computational model.

In summary, the adoption of a WebAssembly subset represents an informed architectural decision that balances theoretical purity with practical engineering concerns. This provides a formally specified, memory-safe, portable execution environment that aligns precisely with the abstract requirements, while gaining access to a mature ecosystem of tools, runtimes, and compiler infrastructure. The subset approach ensures inclusion of only those features strictly necessary for these purposes, maintaining a

minimal trusted computing base and maximal verifiability. The exclusion of unnecessary features (floating-point, 64-bit arithmetic, automatic memory management) keeps the specification tractable for formal analysis while remaining sufficient for the symbolic reasoning and control tasks that characterize autonomous agents. The mandate for specific post-MVP extensions (multi-value returns, bulk memory, multiple memories) reflects a pragmatic recognition that certain carefully designed higher-level constructs provide substantial practical benefits while remaining within the formal verification framework. This foundation positions us to build the additional architectural layers, error handling, IO bridges, and capability management, necessary for a complete agentic execution environment.

4.3 System Architecture

The WebAssembly subset established in the preceding section provides a pure, isolated computational core with rigorously defined semantics. However, this purity comes at a cost: the sandboxed module, by design, possesses no intrinsic mechanism for interacting with the external world or handling runtime failures gracefully. To bridge this computational environment to the practical demands of real-world agentic applications, two critical architectural enhancements are introduced layered atop the base substrate. These components—a principled error-handling protocol and a formal data exchange mechanism—are not modifications to the WebAssembly specification itself but rather carefully designed conventions and host-side mechanisms that preserve the substrate's safety properties while enabling robust, data-intensive computation. A third component integrates the core reasoning capability, the language model, into this architecture, completing the implementation of the agentic primitive $M : \text{String} \to \text{String}$.

4.3.1 Error Handling Protocol

The default error-signaling mechanism in WebAssembly is the *trap*, an abortive termination of the module instance's execution that occurs in response to well-defined error conditions. These conditions include arithmetic errors (division by zero, integer overflow in checked operations), memory access violations (out-of-bounds load or store), invalid table ac-

cesses (indirect call through null or out-of-range table index), call stack exhaustion, and explicit `unreachable` instruction execution. When a trap occurs, the WebAssembly instance immediately halts, control returns to the host with an indication of the trap cause, and the instance's state becomes undefined. From the guest code's perspective, a trap is an unrecoverable event: there is no mechanism for the module to catch, handle, or recover from the error condition within its own execution context.

While this fail-stop behavior is fundamental to WebAssembly's safety guarantees, ensuring that erroneous programs cannot continue executing with corrupted state, it represents an insufficiently expressive error model for building sophisticated applications. Several limitations render traps unsuitable for resilient agentic systems:

1. **Lack of Recoverability**: A trap terminates the entire instance. There is no opportunity for the guest code to perform cleanup, release resources, log diagnostic information, or attempt alternative strategies. This all-or-nothing failure mode is too coarse-grained for agents that must operate robustly in the face of partial failures.

2. **Opacity**: Traps convey minimal information to the host about the nature or context of the failure. While the trap kind (e.g., out-of-bounds memory access) is reported, the semantic meaning, which operation failed, what precondition was violated, what alternative path should be taken, is lost.

3. **Non-Composability**: In a system where multiple operations can fail independently, trap-based error handling forces artificial sequencing. If operation A traps, operation B never executes. There is no mechanism to attempt both operations, collect their individual success/failure statuses, and proceed based on the aggregate result.

4. **Testing Complexity**: Systematically testing error paths in trap-based systems requires elaborate infrastructure to catch traps at the host level and restart execution with modified inputs. This overhead makes comprehensive error-path testing prohibitively expensive.

To address these limitations while preserving the underlying safety guarantees of the substrate, a more expressive *value-based error handling*

paradigm is adopted. This convention, enabled by the WebAssembly multi-value return extension, mandates that any function potentially subject to runtime failure must not trap on error conditions within its control. Instead, it signals success or failure by returning a dedicated error code as part of its result tuple. This approach is inspired by error handling conventions in languages like Go, which explicitly represent errors as first-class values rather than exceptional control flow, and by systems programming practices in C where error codes are used for recoverable errors while signals or aborts are reserved for truly unrecoverable failures.

The protocol operates as follows. For any fallible function, its signature is transformed from:

$$(\text{param}_1, \ldots, \text{param}_n) \to (\text{result}_1, \ldots, \text{result}_m)$$

to the augmented form:

$$(\text{param}_1, \ldots, \text{param}_n) \to (\text{result}_1, \ldots, \text{result}_m, \text{error\_code})$$

where error_code is a `u32` value occupying the final position in the return tuple. The interpretation of this error code follows a strict convention:

- error_code $= 0$ indicates successful execution. All preceding result values in the tuple are valid and semantically meaningful.

- error_code $\neq 0$ indicates a failure condition. The specific non-zero value encodes the error type or cause. All preceding result values in the tuple are undefined and must be discarded by the caller without interpretation.

This binary success/failure indicator is augmented by a standardized error code catalog. Rather than allowing arbitrary error values, a canonical enumeration of error conditions is defined, inspired by the POSIX `errno.h` conventions but tailored for this domain. Representative error codes include:

Table 4.2. Standardized Error Code Catalog

Code	Name	Meaning
0	SUCCESS	Operation completed successfully
1	INVALID_ARGUMENT	Parameter validation failed
2	OUT_OF_RANGE	Index or size exceeds bounds
3	NOT_FOUND	Requested resource does not exist
4	PERMISSION_DENIED	Capability check failed
5	TIMEOUT	Operation exceeded time limit
6	RESOURCE_EXHAUSTED	Memory, quota, or limit reached
7	INTERNAL_ERROR	Host implementation error
8	UNAVAILABLE	External service not reachable

Standardizing these codes ensures interoperability: different host functions and guest modules share a common vocabulary for error conditions, enabling systematic error handling logic that can distinguish between, say, transient failures (TIMEOUT, UNAVAILABLE) that warrant retry and permanent failures (PERMISSION_DENIED, NOT_FOUND) that require alternative strategies.

The practical application of this protocol is illustrated in Figure 4.1, which demonstrates error handling for a key-value store lookup operation. The `kv_get` function returns a tuple $(\text{value\_ptr}, \text{error\_code})$. The caller must first inspect the error code before dereferencing the pointer.

This pattern demonstrates several key properties of the protocol. First, error checking is *explicit* in the control flow: the `if` statement makes the error path visually distinct from the success path, improving code auditability. Second, error handling is *local*: the decision of how to handle an error (propagate it, retry the operation, attempt an alternative) is made by the caller in immediate context, not by a distant exception handler. Third, errors are *composable*: multiple fallible operations can be sequenced, with each operation's error checked independently, and aggregate error handling logic applied based on which operations succeeded or failed.

The protocol effectively emulates a Result$\langle T, E\rangle$ or Either$\langle E, T\rangle$ monad from functional programming, lifting error management from an implicit system-level mechanism (traps) into the explicit value domain of the program. This transformation enables powerful abstractions. For instance, a combinator function `try_sequence` could execute multiple operations,

```
;; Invoke fallible key-value get operation
;; Returns: [value_ptr: u32, error_code: u32]
(call $kv_get (local.get $key_ptr))

;; Multi-value return: stack now contains [value_ptr, error_code]
;; Store error_code first (top of stack)
(local.set $error_code)
;; Store value_ptr second
(local.set $value_ptr)

;; Check error code: if non-zero, enter error handling path
(local.get $error_code)
(if
  (then
    ;; Error path: Log the failure, return null pointer
    ;; and propagate error code
    (call $log_error
      (i32.const 0x1000)  ;; Error message string
      (local.get $error_code))
    (i32.const 0)         ;; Null pointer as result
    (local.get $error_code)
    (return)
  )
)

;; Success path: error_code was 0, safe to use value_ptr
(local.get $value_ptr)
(call $process_value)
;; ... continue normal execution ...
```

Figure 4.1. Value-based error handling in WAT

collecting their error codes, and return success only if all operations succeeded, implementing an atomic transaction pattern purely within guest code.

The relationship between this value-based protocol and the underlying trap mechanism deserves clarification. Traps remain as a backstop for truly unrecoverable errors: programming bugs (out-of-bounds memory access due to incorrect pointer arithmetic), resource exhaustion that cannot be anticipated (call stack overflow), and explicitly unreachable code paths (assertion failures). These represent violations of program correctness rather than expected error conditions. The value-based protocol handles *domain errors*, conditions that are anticipated possibilities in the problem space: a requested key might not exist, a network request might time out, a pars-

ing operation might encounter malformed input. By distinguishing these two error categories, a clean separation is achieved: traps indicate "this should never happen, and if it does, continuation is not possible," while error codes indicate "this is a known possibility, and here is the information needed to decide how to proceed."

This distinction also clarifies the responsibility boundary between host and guest. Host functions are responsible for converting their own internal failures (exceptions in the host language, system call failures) into appropriate error codes before returning to the guest. They must ensure that they never cause a trap in the guest module due to their own implementation errors, for instance by writing out of bounds to guest memory. Conversely, guest code is responsible for checking error codes returned by host functions and handling them appropriately. Failure to check an error code is not a runtime error (the program continues executing with potentially invalid data) but represents a logical bug in the guest code, detectable through static analysis tools that verify error-checking discipline.

The multi-value return extension is essential to this design. Without it, users would be forced to either pack multiple values into memory and return a pointer (inefficient, requiring memory allocation and serialization for every fallible call) or use output parameters (requiring pre-allocated memory, complicating the calling convention). Multi-value returns provide zero-overhead error signaling: the error code is simply an additional register value returned alongside results, with no memory allocation, serialization, or indirection required.

4.3.2 Data Exchange Mechanism

The second architectural challenge concerns data ingress and egress: how does the host provide input data to the sandboxed guest, and how does the guest return complex results to the host? The WebAssembly security model, by design, provides no intrinsic channel for this exchange. A guest module cannot read from standard input, access environment variables, or receive data through any implicit global mechanism. All data must flow through explicit function call boundaries.

For simple scalar values, this presents no difficulty: a host can invoke an exported guest function $f(a_1, \ldots, a_n)$ with numeric arguments that are copied onto the guest's operand stack. However, this mechanism is in-

adequate for the rich, structured data required by agentic applications. Consider the context that must be provided to an agent before it can reason about a task:

- **Task Description**: A potentially long natural language specification of the user's objective, possibly spanning multiple paragraphs.
- **Tool Manifest**: Detailed documentation for each available tool, including signatures, parameter descriptions, example usage, and constraints.
- **Contextual Information**: Relevant background data such as retrieved documents, database query results, sensor readings, or conversation history.
- **String Constants**: Predefined textual templates, error messages, or configuration values needed by the agent's logic.
- **Structured Data**: JSON or XML payloads representing API responses, configuration objects, or complex nested data structures.

None of these can be efficiently represented as a sequence of `u32` arguments. A mechanism is required for transferring arbitrarily large, structured byte sequences between host and guest address spaces, and this mechanism must be both highly efficient (to avoid bottlenecking agent performance) and formally secure (to uphold the sandbox integrity at all times).

The solution, termed the *IO Bridge*, is founded upon two architectural principles: isolated linear memories enabled by WebAssembly's multiple memories extension, and a simple, self-describing binary serialization format. The design operates as follows.

Memory Architecture: The guest module declares two distinct linear memories:

1. **Application Memory** (memory 0): The module's primary working storage, used for its own data structures, local variables spilled to memory, and internal state. This memory is entirely under the guest's control.

2. **IO Memory** (memory 1): A dedicated communication channel for receiving input from and sending output to the host. This memory is conceptually "shared" in that both host and guest can access it, but access is tightly controlled through protocol.

This physical separation provides a strong security boundary. Even if the guest code contains a memory corruption bug that causes it to write to arbitrary addresses within its application memory, it cannot corrupt the IO memory unless it explicitly uses instructions indexed to memory 1. Conversely, the host's writes to IO memory cannot corrupt the guest's application state. This isolation is enforced at the instruction level: `i32.load` defaults to memory 0, while `i32.load` with an explicit memory index 1 accesses IO memory. Static analysis can verify that a module only accesses IO memory in controlled locations, providing assurance that communication follows protocol.

Serialization Format: Data placed in IO memory follows a uniform, length-prefixed binary encoding. Each distinct data element, whether a string, byte array, or structured object, is serialized as:

$$[\text{length} : \texttt{u32}][\text{data} : \text{byte}^{\text{length}}]$$

where the first four bytes encode the element's size in little-endian byte order, followed by that many data bytes. This format is self-describing: a reader can parse the data stream by repeatedly reading a length prefix and then consuming exactly that many bytes. For nested structures, this principle applies recursively. For instance, a list of strings would be encoded as:

$$[\text{count} : \texttt{u32}][\text{len}_1 : \texttt{u32}][\text{str}_1][\text{len}_2 : \texttt{u32}][\text{str}_2] \cdots$$

For variant types or tagged unions, a single-byte is prepended discriminant before the length prefix, allowing up to 256 distinct type tags. This simple scheme is sufficient to represent arbitrary algebraic data types without requiring complex schema definitions or code generation.

This format is chosen for several properties:

- **Binary Safety**: Unlike null-terminated strings, there are no special sentinel values. A string containing null bytes (0x00) is represented

unambiguously.

- **Simplicity**: Parsing requires only primitive operations (read 32-bit integer, advance pointer, copy bytes). No complex grammar or state machine is needed, making it implementable in a few dozen lines of WAT code.

- **Efficiency**: The format is compact (4 bytes overhead per element) and supports zero-copy techniques: a pointer and length into IO memory can reference data in place without copying.

- **Schemaless**: No external schema definition or code generation is required. The format is self-contained and can be parsed generically, with semantic interpretation deferred to application logic.

Alternative serialization formats were considered and rejected for specific reasons. Null-terminated strings (C-style) are unsafe for binary data. Protocol Buffers and similar schema-based formats require complex parsing libraries that would substantially increase the trusted computing base of the guest environment. JSON and XML are textual and verbose, requiring expensive parsing for large documents. MessagePack and similar binary formats are more complex than necessary for this use case. The length-prefixed format represents a sweet spot: sufficiently simple to implement and verify, yet sufficiently expressive to handle the data structures encountered in agentic applications.

Input Protocol: argc/argv Convention: Rather than passing a single monolithic data structure, the host-guest interaction for data input is modeled on the classic UNIX `argc`/`argv` convention used for command-line arguments. This design recognizes that agents typically receive multiple distinct inputs: the primary task description, tool manifests, contextual data, configuration parameters, and so forth. Treating these as separate "arguments" rather than fields in a single structure provides modularity and allows the guest to process inputs selectively.

The protocol operates as follows:

1. **Host Preparation**: Before invoking the guest's main function, the host serializes each input argument using the length-prefixed format and writes the serialized data sequentially into the IO memory. Each

argument occupies a contiguous region, and the host maintains an internal table mapping argument indices to their starting addresses and lengths within IO memory.

2. **Guest Discovery**: The guest begins execution by calling two imported host functions to discover its inputs:
   - `argc`() $\rightarrow$ `u32`: Returns the total number of arguments available.
   - `argv`(n : `u32`) $\rightarrow$ (`u32`, `u32`): Takes an argument index $n \in [0, \texttt{argc}() - 1]$ and returns a tuple (ptr, error_code) where ptr is the address in IO memory (memory 1) of the n-th argument's serialized data, and error_code indicates success (0) or out-of-range access (non-zero).
3. **Guest Parsing**: The guest uses the returned pointer to access the argument data in IO memory. It reads the length prefix, validates it is within bounds, and then interprets the following bytes according to the expected data type. For efficiency, the guest can reference this data in place, or copy it to application memory if it needs to modify it.

This model provides several advantages. First, *lazy evaluation*: the guest need not process all inputs immediately. It can query `argc`(), decide which arguments are needed for the current task, and call `argv`() only for those indices. Second, *clear boundaries*: each argument is an independent unit, preventing parsing errors in one argument from corrupting another. Third, *extensibility*: the host can provide additional arguments (such as debug information or profiling data) by simply increasing `argc`(), and legacy guest code that does not expect these arguments will simply ignore them.

The `argv`() function is designed to never trap. If called with an invalid index ($n \geq \texttt{argc}()$), it returns a null pointer (0) and a non-zero error code, allowing the guest to handle this gracefully. This design follows the value-based error handling protocol.

Output Protocol: Result Vector: For returning complex results, the IO Bridge provides a symmetric mechanism. The guest module prepares its output data in application memory (memory 0) using the same

length-prefixed serialization format. It then invokes an imported host function: `resv`(ptr : `u32`) → (`u32`, `u32`) which takes a pointer to the serialized data in application memory and returns a result ID and error code. The host function performs the following steps:

1. Safely reads the length prefix from the guest's application memory, validating it does not exceed reasonable bounds.

2. Copies the specified number of bytes from guest memory to a host-managed buffer.

3. Increments an internal result counter, assigning a unique ID to this result.

4. Returns the result ID and success code (0) to the guest.

The guest can call `resv`() multiple times during a single invocation to return multiple distinct results, for instance, intermediate outputs, final answer, and diagnostic logs. The host maintains these results in order, associating each with its result ID. After the guest's main function returns, the host can retrieve all accumulated results.

The security properties of this design are critical. The guest passes a pointer to its *own* memory (memory 0), not to shared IO memory. The host function performs a *copy* from guest memory to host memory, not a direct reference. This ensures the host receives a stable snapshot of the data: even if the guest continues executing and modifies the memory region after the `resv`() call, the host's copy remains unchanged. Conversely, the guest never receives a pointer into host memory, preventing it from reading or corrupting host state.

String Constants and Shared Data: A special case of data exchange concerns *constants*, read-only string or data values that the guest code needs to reference frequently. Examples include:

- HTTP method strings ("GET", "POST", "PUT", "DELETE")

- Standard error messages ("Invalid argument", "Resource not found")

- AI system prompts ("Summarize the following text:", "Translate to French:")

- Configuration keys ("max_retries", "timeout_ms")

Rather than having the guest embed these strings as data segments (consuming space in the guest's memory and requiring initialization), or having the host pass them as arguments (redundantly transmitting the same data on every invocation), they are provided as *host functions returning constant pointers*. For instance: `http.GET()` → `u32` This function, when called, returns a pointer to the string "GET" in IO memory (memory 1). The host has pre-initialized IO memory with these constants at fixed offsets, and the function simply returns the appropriate address. This approach has several benefits:

- **Zero Cost**: Calling the function is effectively free (a constant return), and the string data is shared across all invocations.
- **Type Safety**: Each constant is accessed through a named function, making code more readable and preventing typos.
- **Capability Control**: By only exposing certain constant functions, the host can control what strings the guest can reference, useful for restricting API access.

Similarly, functions are provided for obtaining pointers to structured configuration data or lookup tables that the guest needs to reference but should not modify.

Bidirectional Communication Summary: The complete IO Bridge architecture thus provides three channels:

1. **Input Channel** (Host → Guest): Via `argc()`/`argv()`, the guest discovers and accesses serialized arguments in IO memory.
2. **Output Channel** (Guest → Host): Via `resv()`, the guest sends serialized results from its application memory to the host.
3. **Constant Channel** (Host → Guest): Via named constant functions (e.g., `http.GET()`), the guest obtains pointers to shared read-only data in IO memory.

All three channels respect the sandbox boundary: the guest never accesses host memory directly, and the host never allows the guest to corrupt

its own state. The use of two separate memory spaces ensures that even a buggy or malicious guest cannot exploit the communication channel to escape the sandbox. The protocol is also efficient: most operations involve copying pointers rather than data, and bulk data transfer uses the optimized `memory.copy` instruction or direct pointer references.

4.3.3 Reasoning Integration

With the error-handling protocol and IO Bridge in place, the integration of the core reasoning capability—the language model M—into the substrate is now addressed. Recall the fundamental agentic primitive: $M : \text{String} \to \text{String}$ This primitive maps a contextual prompt to a generated response. Implementing this primitive *within* the WebAssembly guest would violate the architectural principles: it would require the guest to possess networking capabilities (to contact an LLM API), cryptographic capabilities (to handle TLS), and substantial computational resources (if running a local model). These capabilities would dramatically expand the trusted computing base and contradict the principle of minimal guest authority.

Instead, the M function is realized as an imported host function, callable by the guest but implemented entirely in the privileged host environment. This design maintains a clean separation: all side effects, network I/O, GPU computation, and interaction with external LLM services, are encapsulated within the host, while the guest retains only pure computational logic for constructing prompts and parsing responses.

Host Function Interface: The core reasoning function is exposed to the guest as: $\texttt{ai.query}(\text{prompt\_ptr} : \texttt{u32}, \text{config\_ptr} : \texttt{u32}) \to (\texttt{u32}, \texttt{u32})$ The parameters are:

- prompt_ptr: Pointer to a length-prefixed string in the guest's application memory (memory 0) containing the prompt text.
- config_ptr: Pointer to a length-prefixed configuration structure specifying model selection, temperature, maximum tokens, and other inference parameters. A null pointer (0) indicates default configuration.

The return value is a tuple $(\text{response\_ptr}, \text{error\_code})$ following the error-handling protocol, where response_ptr points to the serialized response

string in IO memory (memory 1), and error_code indicates success or the nature of any failure (timeout, invalid request, model unavailable).

Operational Flow: The lifecycle of a reasoning call proceeds as follows:

1. **Prompt Construction**: The guest's logic determines that it needs to consult the language model. It constructs a prompt string in its application memory, potentially by concatenating multiple components: task description, relevant context, tool documentation, and specific questions.

2. **Function Invocation**: The guest calls `ai.query`(), passing a pointer to this prompt and optionally a configuration structure. Control transfers from the guest to the host runtime.

3. **Host Processing**: The host function, implemented in the host language (e.g., Rust), uses its privileged access to the guest's memory to safely read the prompt data. It validates the length prefix, allocates a host-side buffer, and copies the prompt text. It then performs the necessary side effects:

   - Establishes a network connection to the LLM API endpoint (e.g., OpenAI, Anthropic, or a self-hosted model server).
   - Formats an API request according to the service's specification, including the prompt, inference parameters, and authentication credentials.
   - Sends the request and awaits the response, potentially with timeout handling.
   - Parses the API response to extract the generated text.

4. **Result Serialization**: The host serializes the response string using the length-prefixed format and writes it to a region of the IO memory (memory 1). It obtains a pointer to this region.

5. **Return to Guest**: The host function returns the tuple $(\text{response\_ptr}, 0)$ to the guest if successful, or $(0, \text{error\_code})$ if an error occurred.

6. **Response Processing**: Control returns to the guest's WAT code. The guest immediately inspects the error code. If zero, it uses the

response pointer to access the generated text in IO memory and proceeds with its logic. If non-zero, it enters an error-handling path, potentially logging the failure, retrying with a modified prompt, or returning an error to its own caller.

This flow exemplifies the architectural separation of concerns: pure reasoning logic (prompt construction, response parsing, control flow) resides in the verifiable guest code, while impure, platform-specific, security-sensitive operations (network communication, API authentication, error handling) are entirely encapsulated in the auditable host implementation.

Variant Functions: To support common usage patterns, several specialized variants of the core reasoning function:

- `ai.assist`(system_ptr, user_ptr, config_ptr) → (`u32`, `u32`): Takes both a system instruction (meta-prompt defining the model's role) and a user query, formatting them appropriately for chat-based APIs that distinguish system and user messages.

- `ai.stream`(prompt_ptr, callback_fn, config_ptr) → `u32`: For long responses, streams generated tokens incrementally by invoking a guest-provided callback function for each token, enabling real-time display or early termination.

- `ai.batch`(prompts_array_ptr, count, config_ptr) → (`u32`, `u32`): Submits multiple prompts as a batch to leverage parallelism in the LLM backend, returning an array of response pointers.

Additionally, constant functions are exposed returning pointers to commonly used system instructions:

- `ai.SUMMARIZE`(): Returns pointer to "Provide a concise summary of the following text:"

- `ai.TRANSLATE`(): Returns pointer to "Translate the following text to [language]:"

- `ai.CODE_REVIEW`(): Returns pointer to "Review the following code for bugs and suggest improvements:"

These constants provide convenient access to standard prompts while allowing customization when needed.

Security Considerations: The reasoning integration introduces several security considerations. First, *prompt injection*: a malicious or compromised guest could construct prompts designed to manipulate the LLM into revealing information, ignoring instructions, or generating harmful content. The host must implement robust prompt sanitization, input validation, and output filtering to mitigate these risks. Second, *resource exhaustion*: unconstrained reasoning calls could exhaust API quotas or incur unbounded costs. The host should enforce rate limits, token budgets, and timeout constraints per guest instance. Third, *information leakage*: the prompts sent to external LLM services may contain sensitive information. The host must ensure appropriate data handling policies, potentially using local models for sensitive contexts or anonymizing prompts before external transmission.

These security mechanisms are implemented entirely in the host layer, transparent to the guest. From the guest's perspective, `ai.query`() is simply a function that sometimes succeeds and sometimes fails with an error code. The guest need not, and indeed cannot, concern itself with how the host implements rate limiting, prompt filtering, or service selection. This encapsulation maintains the simplicity and verifiability of guest code while concentrating security policy enforcement in the auditable host implementation.

Configuration and Model Selection: The configuration structure passed to reasoning functions allows fine-grained control over inference parameters:

- **Model Selection**: An enumeration or string identifying which LLM to use (e.g., "gpt-4", "claude-3-sonnet", "local-7b").
- **Temperature**: Controls randomness in generation, with higher values producing more creative but less deterministic outputs.
- **Max Tokens**: Limits response length to prevent excessively long generations.
- **Stop Sequences**: Strings that, when generated, terminate the response early.

- **Top-p/Top-k**: Nucleus sampling parameters for controlling output diversity.

By exposing these parameters to the guest through the configuration structure, sophisticated prompting strategies where the agent dynamically adjusts inference parameters based on context. For instance, using low temperature for factual queries and high temperature for creative tasks, or selecting smaller, faster models for simple sub-tasks and larger models for complex reasoning.

The reasoning integration thus completes the implementation of the agentic primitive M within this substrate framework. The guest code can invoke the language model as easily as calling any other function, while the architectural layering ensures that all complex, security-sensitive operations remain within the trusted host environment. This design enables the programmatic composition of reasoning steps, supporting advanced patterns like iterative refinement, self-critique, and tree-of-thought exploration, as will be demonstrated in the subsequent section on implementation methodology.

4.4 Reference Implementation

With the formal substrate defined, the WebAssembly instantiation specified, and the architectural enhancements established, a concrete reference implementation is now presented demonstrating the practical viability of the framework. This implementation serves multiple purposes: it validates that the theoretical design can be realized with existing technology, it provides a concrete artifact for evaluation and experimentation, it establishes implementation patterns and best practices for developers, and it identifies pragmatic considerations not evident from the abstract specification alone. While the principles articulated thus far are deliberately language-agnostic to ensure portability, this reference implementation makes specific technology choices to ground the discussion in working code.

Technology Stack Selection: The reference implementation utilizes Wasmtime as the WebAssembly runtime engine, a production-grade implementation from the Bytecode Alliance recognized for its performance, security, and standards compliance. Wasmtime is implemented in Rust

and provides both a command-line interface for standalone execution and a comprehensive embedding API for integration into host applications. Its selection is motivated by several factors: rigorous adherence to the WebAssembly specification with extensive conformance testing, support for all required post-MVP extensions (multi-value returns, bulk memory, multiple memories), active maintenance and rapid incorporation of emerging standards, high-quality documentation and examples, and proven production deployment in systems ranging from edge computing platforms to cloud-native infrastructure.

The host environment, the privileged layer that embeds the Wasmtime runtime and implements the host functions exposed to guest modules, is implemented in Rust. This choice leverages Rust's memory safety guarantees to build a secure embedder that cannot introduce memory corruption vulnerabilities even when interfacing with low-level WebAssembly memory operations. Rust's ownership system ensures that host-side pointers to guest memory are properly validated and cannot outlive their validity period, preventing use-after-free errors. The language's zero-cost abstractions enable high-performance implementations of the IO Bridge and error handling mechanisms without sacrificing safety.

Agent logic, the code executing within the WebAssembly sandbox, can be authored in any language that compiles to the WebAssembly target. For the reference implementation, examples are provided in multiple source languages to demonstrate portability. Primary examples use Rust compiled with the `wasm32-unknown-unknown` target, leveraging its excellent WebAssembly toolchain support and ability to generate small, efficient binaries. Supplementary examples demonstrate C compilation via Emscripten (with WASI features explicitly disabled to maintain the minimal capability model), Go compilation via TinyGo (a Go compiler optimized for embedded systems and WebAssembly), and AssemblyScript (a TypeScript-like language designed specifically for WebAssembly generation). Additionally, WAT examples are provided (WebAssembly Text) examples for scenarios requiring direct control over instruction sequences or for pedagogical purposes where seeing the low-level representation aids understanding.

Host Implementation Architecture: The host implementation is structured as a modular system with clearly delineated responsibilities.

At its core is the `Runtime` struct, which encapsulates a Wasmtime `Engine` (the compilation and execution engine) and a `Linker` (the mechanism for registering and resolving host functions). The runtime maintains global configuration including timeout limits, memory quotas, and security policies that apply uniformly to all guest modules.

Module instantiation proceeds through a structured pipeline. First, the module's binary is loaded from disk or received as a byte stream. Second, Wasmtime's validator performs comprehensive static analysis, verifying type correctness, stack discipline, control flow validity, and resource bounds. Invalid modules are rejected before any instantiation occurs. Third, for valid modules, the runtime creates a `Store`, Wasmtime's representation of the instantaneous execution state, and links the module's imports to host-provided implementations registered in the `Linker`. Finally, the module is instantiated, allocating its linear memories, initializing globals and tables, and executing any start function.

The `Linker` serves as the registry for the standard library of host functions detailed in Table 4.3. Each host function is implemented as a Rust function with a signature compatible with the WebAssembly type system, using Wasmtime's procedural macros to generate the necessary marshaling code. The linking process ensures type safety: if a guest module imports a function f with signature $(i32, i32) \to i32$, and the host provides an implementation with a different signature, the instantiation fails with a clear type mismatch error.

Table 4.3. Standard Library Host Functions

Function(s)	Description
System	
sys.argc	Number of input arguments.
sys.argv	Argument by index.
sys.resv	Save output result.
sys.nil	Null pointer constant (0).
sys.alloc	Allocate N bytes in shared memory.
dbg.inspect	Print raw u32 value.
dbg.trace	Print N bytes with length prefix.
dbg.log	Print pointer as string.
String Manipulation	
str.clone, str.slice	Copy and slice operations.
str.compare, str.eq	Comparison operations.
str.index_of, str.contains, str.starts_with, str.ends_with	Substring search operations.
str.trim, str.to_upper	Whitespace and case operations.
str.split, str.join	Divide and combine strings.
str.tag, str.untag	Metadata tag operations.
Filesystem	
fs.read, fs.write, fs.append, fs.rm	Sandboxed file operations.
HTTP	
http.request	Generic HTTP request.
http.serve	Serve content on port.
http.GET, http.POST, http.PATCH, http.DELETE	HTTP method constants.
AI	
ai.query	Query AI model.
ai.assist	Query with system instruction.
ai.SUMMARIZE, ai.TRANSLATE	AI instruction constants.
Key-Value Store	
kv.get, kv.set	Retrieve and store by key.

Memory Management: Guest modules require memory management for dynamic allocation, as static data segments are insufficient for complex computations. The reference implementation provides a simple but effective bump allocator exposed through the `sys.alloc` host function. This allocator maintains a single `u32` global variable representing

the "bump pointer", the address of the next available byte in the guest's application memory (memory 0).

Allocation proceeds as follows: when the guest calls `sys.alloc`(n) requesting n bytes, the allocator first checks whether sufficient space remains between the current bump pointer and the end of allocated memory. If the requested allocation would exceed the current memory size, the allocator invokes the WebAssembly `memory.grow` instruction to request additional memory pages from the runtime. This operation may fail if the memory has reached its declared maximum size or if the host enforces a quota. On failure, `sys.alloc` returns 0 (null pointer) and sets an error code. On success, the allocator returns the current bump pointer value and increments it by n bytes (rounded up to maintain alignment).

This bump allocation strategy provides constant-time allocation with minimal bookkeeping overhead. Its primary limitation is the absence of deallocation: once allocated, memory cannot be reclaimed. For the episodic execution model employed by agents, where each task invocation creates a fresh module instance that is discarded upon completion, this limitation is acceptable. The entire memory space is implicitly "deallocated" when the instance terminates. For long-lived agent instances requiring memory reclamation, the allocator can be augmented with a simple free list or replaced with a more sophisticated allocator like dlmalloc ported to WebAssembly.

IO Bridge Implementation: The input and output protocols are implemented through coordinated host and guest components. On the host side, a `ContextManager` struct prepares input arguments before guest invocation. For each input datum (task description, tool manifest entry, contextual document), the manager serializes it using the length-prefixed format into a contiguous byte buffer. These buffers are written sequentially into the guest's IO memory (memory 1), and the manager records each argument's starting address and length in an internal table indexed by argument number.

The `argc`() and `argv`(n) host functions query this table. The `argc`() implementation simply returns the table's length. The `argv`(n) implementation validates that n is within bounds, returning an error code if not, and otherwise returns a tuple containing the pointer to the n-th argument in IO memory and a success code.

On the guest side, the reference implementation provides a `libstd` library (compiled to WebAssembly from Rust or C) that wraps these raw host functions with ergonomic APIs. The library's `get_arg` function calls `argv`(n), checks the error code, and if successful, reads the length prefix from IO memory and constructs a safe slice or string reference. This abstraction hides the low-level pointer manipulation from application logic, reducing the likelihood of memory safety errors.

For output, the `resv`() host function implementation uses Wasmtime's memory introspection API to safely read from the guest's application memory. Given a pointer provided by the guest, the function first validates that the pointer lies within the bounds of memory 0, preventing out-of-bounds access. It then reads the 4-byte length prefix, validates that the length is reasonable (e.g., less than 100MB to prevent denial-of-service via memory exhaustion), and copies the specified bytes into a host-allocated buffer. This buffer is stored in a vector of results associated with the current invocation, and the function returns the result's index.

Error Handling Utilities: To facilitate the adoption of the value-based error handling protocol, the reference implementation provides utility functions and macros on both host and guest sides. On the host side, a `HostResult` $<$ `T` $>$ type wraps the result of host function operations. This type can be automatically converted to the $(value, error\_code)$ tuple required by the WebAssembly calling convention, with the `Ok`(`v`) variant mapping to $(v, 0)$ and `Err`(`e`) variant mapping to $(0, \text{error\_code})$ where the error enum is converted to the standardized error code.

On the guest side, macros such as `try_call!` automate the pattern of calling a fallible function, checking its error code, and either propagating the error or proceeding with the result. For example:

```
let value = try_call!(kv_get(key_ptr),
                      "Failed to retrieve key");
```

This macro expands to code that calls `kv_get`, checks the error code, and if non-zero, logs the error message and returns immediately, propagating the error to the caller. This reduces boilerplate and makes error handling paths explicit and auditable.

Tool Discovery and Dynamic Invocation: A critical component of the reference implementation is the `ToolRegistry`, a host-side subsys-

tem responsible for discovering available tools and making them known to guest modules through the contextualization function ρ. At runtime initialization, the registry introspects the `Linker` to enumerate all registered host functions. For each function, it retrieves metadata including the function name, type signature (parameters and return types), and associated documentation comment parsed from the source code.

This metadata is formatted into a structured tool manifest document, a JSON or YAML representation listing each tool with its signature, parameter descriptions, and usage examples. The manifest is serialized and passed to the guest module as one of the initial arguments accessible via `argv`(). The guest's prompt construction logic parses this manifest and incorporates it into the system prompt sent to the language model, implementing the in-context learning approach to tool awareness.

For example, the manifest entry for `http.request` might include:

```
{
  "name": "http.request",
  "signature": "(method_ptr, url_ptr, body_ptr,
                config_ptr) -> (response_ptr, error_code)",
  "parameters": {
    "method_ptr": "Pointer to HTTP method string (GET, POST, etc.)",
    "url_ptr": "Pointer to URL string",
    "body_ptr": "Pointer to request body, or 0 for GET",
    "config_ptr": "Pointer to config struct, or 0 for defaults"
  },
  "returns": "Tuple of (response_ptr, error_code).
              On success, response_ptr points to response data.",
  "example": "call $http.request ($http.GET) ($url)
                    (i32.const 0) (i32.const 0)"
}
```

This rich metadata enables the language model to generate correct function invocations without requiring fine-tuning on the specific tool set, demonstrating the power of in-context learning for capability-aware code generation.

Reasoning Model Integration: The `ai.query` family of functions is implemented as host functions that bridge to external language model

APIs. The reference implementation provides a pluggable `LLMProvider` trait that abstracts the specifics of different API providers (OpenAI, Anthropic, local inference servers). Each provider implementation handles API-specific details such as request formatting, authentication, rate limiting, and response parsing.

The generic `ai.query` host function implementation proceeds as follows:

1. Safely read the prompt string from the guest's application memory using the provided pointer.
2. Parse the configuration structure (if provided) to extract model selection, temperature, and other parameters.
3. Select the appropriate `LLMProvider` based on the requested model.
4. Invoke the provider's `complete` method, passing the prompt and configuration, which performs the network request and awaits the response.
5. Handle provider-specific errors (timeouts, rate limits, invalid requests) by mapping them to standardized error codes.
6. On success, serialize the response text using the length-prefixed format, write it to IO memory, and return the pointer and success code.

For local model execution, the reference implementation includes an `LLMProvider` that interfaces with inference engines like llama.cpp or Candle (a Rust ML framework), loading quantized models and performing inference on CPU or GPU. This path avoids network latency and enables offline operation, though with higher host-side computational requirements.

Execution Model and Lifecycle: The operational model for executing agent programs follows a lightweight, ephemeral pattern inspired by serverless computing. Each task invocation creates a fresh module instance from the compiled WebAssembly binary. The host prepares the execution context by serializing inputs and writing them to IO memory, then invokes the module's exported `run` function with signature `void` $\rightarrow$ `u32`. This function serves as the agent's entry point, where the return value is an error code (0 for success, non-zero for failure).

The agent's `run` function orchestrates the complete problem-solving process: it queries `argc` and `argv` to discover inputs, constructs an initial prompt incorporating task description and tool manifest, invokes `ai.query` to generate a plan or response, parses the model's output to extract tool invocations, executes those tools through imported host functions, potentially iterates with additional reasoning calls based on intermediate results, and finally calls `resv` to return the ultimate result.

If the `run` function returns a non-zero error code, indicating task failure, the host interprets this as a signal to retry. The retry mechanism captures any diagnostic output written to the debug logging functions (`dbg.log`, `dbg.trace`), appends this information to the original task description as feedback, and creates a new module instance with the augmented context. This retry loop has a configurable maximum iteration count to prevent infinite loops, and each iteration operates on a fresh instance to avoid state corruption from the previous failed attempt.

This "let it crash" philosophy, borrowed from Erlang's supervision trees, treats the WebAssembly instance as a lightweight, disposable computational unit. Failed instances are simply discarded, and their allocated resources (memory, file handles held by the host) are automatically reclaimed. This approach has several advantages: it eliminates the need for complex state management and error recovery within the guest code, it prevents error propagation where a partially executed task corrupts state used by subsequent tasks, and it provides natural isolation between attempts, ensuring that a bug triggered in one iteration cannot affect the next.

Example: Self-Refinement Loop: To demonstrate the framework's expressiveness, Figure 4.2 presents a complete WAT implementation of an iterative self-refinement reasoning pattern. This program accepts an initial query as input, generates an initial response, then repeatedly invokes the reasoning model in a "critique and refine" mode, using each iteration's output as input to the next.

```
(func (export "run") (result i32)
(local $error i32)
(local $query_ptr i32)
(local $response_ptr i32)
(local $refine_prompt_ptr i32)
(local $iterations i32)

;; Get initial query from argv[0]
(call $argv (i32.const 0))
(local.set $error)
(local.set $query_ptr)
(if (local.get $error)
  (then (return (local.get $error)))
)

;; Get refinement prompt template
(local.set $refine_prompt_ptr (call $ai_REFINE))

;; Set iteration count
(local.set $iterations (i32.const 3))

;; Refinement loop
(loop $refine_loop
  (if (i32.eq (local.get $iterations) (i32.const 0))
    (then (return (i32.const 0)))  ;; Success
  )

  ;; Query AI with current input
  (call $ai_query
    (local.get $refine_prompt_ptr)
    (local.get $query_ptr))
  (local.set $error)
  (local.set $response_ptr)

  ;; Check for error
  (if (local.get $error)
    (then (return (local.get $error)))
  )

  ;; Log intermediate result
  (call $log (local.get $response_ptr))

  ;; Use response as input for next iteration
  (local.set $query_ptr (local.get $response_ptr))

  ;; Decrement counter and loop
  (local.set $iterations
    (i32.sub (local.get $iterations) (i32.const 1)))
  (br $refine_loop)
)

(i32.const 0)  ;; Success
)
```

Figure 4.2. Iterative self-refinement in WAT

This example illustrates several framework features: explicit error checking after each fallible operation, use of constant functions (`ai.REFINE`) for standard prompts, structured control flow with labeled loops and conditional branches, and the ability to compose multiple reasoning calls programmatically. The entire program is deterministic and verifiable: its control flow is statically analyzable, its resource usage is bounded (fixed number of iterations), and its interaction with the host is limited to the explicitly imported functions.

Interoperability and Portability Validation: A key validation of the framework's design is demonstrating true portability: that the same compiled WebAssembly module can execute without modification across different host implementations and platforms. The reference implementation includes a secondary host environment implemented in TypeScript using the Wasmtime npm bindings, targeting Node.js as the runtime. This TypeScript host implements the identical host function interface (Table 4.3) as the Rust host, with function names, signatures, and error codes matching exactly.

A test suite of agent modules, compiled once from various source languages, is executed on both the Rust host (Linux x86-64), the Rust host (macOS ARM64), and the TypeScript host (Node.js on Windows x86-64). All tests produce identical results across these configurations, validating that the WebAssembly abstraction successfully isolates agent logic from platform-specific details. This portability extends to different runtime engines: the same modules execute correctly using Wasmtime, Wasmer, and the browser's native WebAssembly engine (with a browser-compatible host interface), demonstrating that the framework is not tied to a specific runtime implementation.

Development Workflow and Tooling: The reference implementation includes development tools to streamline agent creation. A command-line tool, `wat − agent`, provides scaffolding for new agent projects, generating boilerplate import declarations and stub functions. An integrated testing framework enables unit testing of individual agent functions by mocking host function responses, allowing logic to be validated without requiring full end-to-end execution. A debugging proxy allows stepping through WAT code instruction-by-instruction, inspecting memory and local variables at each step.

For developers preferring higher-level languages, the framework provides SDK libraries in Rust, C, and AssemblyScript that wrap the raw host function imports with ergonomic, type-safe APIs. These SDKs handle the details of length-prefixed serialization, error code checking, and memory management, allowing developers to focus on agent logic rather than low-level details. For example, the Rust SDK provides:

```
let query = get_arg(0)?;  // Get first argument, propagate errors
let response = ai::query(&query, Default::default())?;
save_result(&response)?;  // Return result to host
Ok(())  // Indicate success
```

This high-level code compiles to efficient WebAssembly that interacts correctly with the host through the defined protocols, demonstrating that the framework's low-level rigor does not preclude developer-friendly abstractions.

Performance Characteristics: Preliminary benchmarking of the reference implementation reveals performance characteristics suitable for interactive agentic applications. Module instantiation from a cached compiled binary takes approximately 50-200 microseconds depending on module size, making the ephemeral execution model practical for high-frequency invocations. The overhead of the capability-based security model, crossing the host-guest boundary for each tool invocation, is measured at 1-5 microseconds per call, negligible compared to the latency of actual tool operations (milliseconds for I/O operations, seconds for AI reasoning). Memory overhead per module instance is proportional to declared memory size, typically 1-10 MB for agent modules, enabling thousands of concurrent instances on a single server.

The error handling protocol introduces zero runtime overhead for the success path (error code is simply an additional return value in a register) and minimal overhead for the error path (an extra conditional branch). The IO Bridge's length-prefixed serialization is highly efficient, with throughput limited primarily by memory bandwidth rather than CPU processing. For a 1 MB document passed as input, serialization and copying take approximately 100 microseconds, demonstrating the practicality of transferring substantial context to agents.

4.5 Evaluation and Applications

Having established the theoretical foundations, practical implementation, and reference architecture of the framework, a systematic evaluation is now presented demonstrating its performance characteristics, true platform portability, and security properties. The evaluation is structured around three complementary experiments: comparative performance analysis against conventional agent execution environments, validation of cross-platform behavioral determinism, and analytical assessment of the attack surface area relative to ambient authority execution models.

4.5.1 Computational Efficiency Analysis

To quantify the performance characteristics of the framework relative to conventional agent deployment approaches, two representative tasks: Levenshtein edit distance computation, a CPU-bound dynamic programming algorithm commonly used for fuzzy string matching and similarity analysis, and LZW string compression, a dictionary-based compression algorithm representative of data processing workloads encountered in agentic systems.

Experimental Setup: For each task, three versions were implemented representing different deployment strategies. The *native Python* implementation uses idiomatic Python 3.12 with standard library features, representing the most common approach for rapid agent prototyping and development. The *containerized Python* version executes the identical Python code within a Docker container using the official `python:3.12-slim` base image, representing a common production deployment pattern that provides process isolation while maintaining language runtime convenience. The *framework* version executes within the Rust-based host environment using Wasmtime 28.0 as the WebAssembly runtime. For the Levenshtein distance task, the algorithm is implemented directly in WAT, demonstrating pure guest-side computation. For the LZW compression task, the WAT module delegates the core compression logic to a high-performance host function implemented in Rust, demonstrating the framework's capability for efficient host-guest collaboration.

Each implementation was executed across a range of input sizes to characterize scaling behavior and identify performance trends. For Leven-

shtein distance, string pairs were tested ranging from 10 characters to 1,000 characters in length, sampling at logarithmically spaced intervals to capture both small-input overhead characteristics and large-input asymptotic behavior. For LZW compression, input strings were tested ranging from 100 to 5,000 characters, again using logarithmic sampling. All benchmarks were executed on identical hardware (AMD Ryzen 9 5950X, 64GB RAM, Ubuntu 22.04 LTS) to eliminate platform-dependent variability, with 50 independent trials per configuration to account for measurement variance and enable statistical analysis. Execution time was measured using the `hyperfine` benchmarking tool, which provides robust statistical analysis with outlier detection and warmup runs. Peak memory usage was profiled using `/usr/bin/time` with the `-v` flag, which reports maximum resident set size from kernel accounting.

Levenshtein Distance Results: The Levenshtein distance benchmark, visualized in Figure 4.3, reveals substantial performance advantages for the framework across all input sizes. The visualization uses a logarithmic scale on the y-axis to accommodate the wide range of execution times, spanning from single-digit milliseconds for small inputs to hundreds of milliseconds for large inputs.

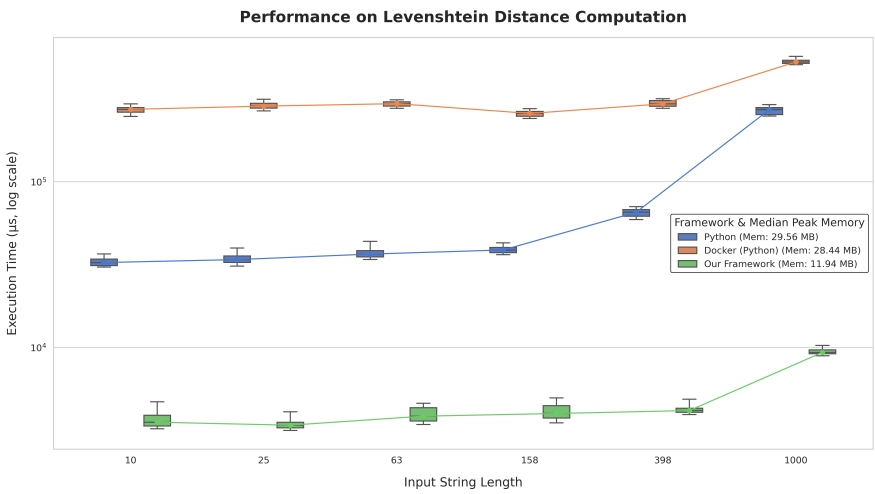


Figure 4.3. Levenshtein Distance Computation Performance

For small inputs (10-character strings), native Python completes in a median of 32.6 milliseconds. This baseline includes Python interpreter startup overhead, module loading, and the actual computation. Containerized Python exhibits significantly higher latency, requiring 273 milliseconds, with the additional overhead arising from Docker container initialization, namespace setup, and cgroup configuration. The framework completes the identical computation in 3.5 milliseconds, representing a 9.3× speedup over native Python even for these short-duration tasks where startup overhead might be expected to dominate.

As input size increases, the performance gap widens consistently. For medium-sized inputs (158-character strings), Python requires a median of 38.8 milliseconds, Docker 258 milliseconds, and this framework 4.1 milliseconds, representing a 9.5× speedup over native Python. The framework's advantage here reflects the elimination of interpreter dispatch overhead in the dynamic programming algorithm's inner loops. For the largest tested inputs (1,000-character strings), the disparity becomes most pronounced: Python's median execution time reaches 272 milliseconds, Docker 528 milliseconds, while this framework completes in 9.5 milliseconds. This 28.6× speedup indicates that the framework's architectural advantages compound with problem size, as the proportional cost of the interpreter's per-operation overhead increases with algorithmic complexity.

Memory consumption patterns strongly reinforce these performance findings. Native Python maintains a median peak memory footprint of 30.2 MB across all input sizes, reflecting the Python interpreter's base memory requirements (runtime, loaded modules, standard library) plus the allocation overhead for string objects and the dynamic programming matrix. Docker exhibits a slightly lower footprint at 29.1 MB, as the container's minimal base image reduces some ancillary memory usage. The framework, by contrast, exhibits a median peak memory of only 12.2 MB, representing a 2.5× memory reduction compared to native Python. This dramatic improvement reflects the minimal footprint of the Wasmtime runtime (approximately 2 MB baseline) combined with the explicit, unboxed memory layout of the WebAssembly linear memory model, which avoids the per-object metadata overhead inherent in Python's reference-counted heap.

LZW Compression Results: The LZW compression benchmark,

shown in Figure 4.4, demonstrates the framework's ability to accelerate workloads through strategic delegation of performance-critical operations to efficient host-side implementations. This pattern is representative of real-world agentic systems where computationally intensive primitives (parsing, serialization, cryptography) are best implemented in compiled languages while control flow and reasoning remain in the flexible agent layer.

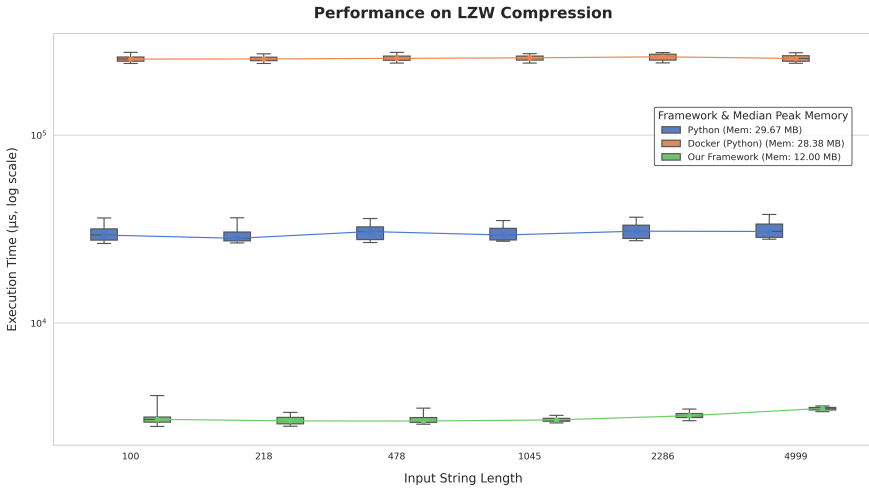


Figure 4.4. LZW String Compression Performance

The results exhibit a remarkably consistent performance profile across input sizes. For inputs ranging from 100 to 5,000 characters, this framework maintains median execution times between 3.0 and 3.5 milliseconds, with minimal variance as indicated by the tight interquartile ranges in the box plots. This near-constant execution time reflects the fact that the WAT module's role is merely to marshal arguments and invoke the host function, with the actual compression performed in optimized Rust code. Native Python requires 29 to 32 milliseconds across the same input range, representing a consistent 9-10× speedup for the framework. Python's performance is relatively flat because the standard library's LZW implementation is itself written in C (the `zlib` module), though the function call overhead and object marshaling still impose measurable costs.

Docker containerization introduces substantial overhead for this workload, with median times ranging from 250 to 265 milliseconds. This overhead, which dwarfs the actual computation time, arises from the container lifecycle: for each invocation, Docker must create a new container, mount volumes, configure networking, execute the Python script, and cleanup. This demonstrates that containerization, while providing isolation, introduces latency that makes it impractical for short-lived computational tasks typical of agentic workflows.

Memory usage patterns closely parallel the Levenshtein results. Python maintains a median footprint of 30.3 MB, Docker 29.1 MB, and this framework 12.2 MB. The framework's advantage is again attributable to the lightweight nature of the Wasmtime runtime and the minimal memory required by the guest module, which in this case serves primarily as a thin marshaling layer rather than implementing the compression algorithm itself.

Performance Analysis: The observed speedups, ranging from 9× to 29× depending on workload characteristics, arise from several complementary architectural factors. First, for the Levenshtein distance task where the algorithm is implemented entirely in guest WAT code, the ahead-of-time compilation pipeline (WAT to WebAssembly bytecode, followed by Wasmtime's optimizing JIT compilation to native machine code) produces instruction sequences that avoid interpreter dispatch overhead entirely. The resulting native code executes with tight inner loops comparable to hand-written assembly, with no per-iteration function call overhead or dynamic type checking.

Second, for the LZW compression task, the framework's architecture enables strategic performance optimization through capability delegation. The computationally intensive compression logic executes in pre-compiled Rust code, bypassing both Python's interpreter overhead and WebAssembly's sandboxing boundary checks for the performance-critical path. The WAT guest code serves only to orchestrate the operation, a pattern that combines the flexibility of scriptable control flow with the raw performance of systems programming.

Third, the minimal runtime footprint of WebAssembly contributes measurable advantages. Wasmtime's baseline memory overhead is approximately 2 MB, compared to Python's 25-30 MB interpreter infrastructure.

This difference compounds in memory-constrained environments or scenarios requiring concurrent execution of multiple agent instances, where the framework enables substantially higher density.

Critically, these performance advantages coexist with the security properties demonstrated in Section 4.5.3. The same modules achieving 9-29× speedups execute within a formally verified sandbox with a drastically reduced attack surface, a 95% reduction in exposed system calls. This contradicts the common assumption of a "security tax" where stronger isolation necessarily implies performance degradation. The framework demonstrates that principled architectural design, grounded in capability-based security and formal semantics, can simultaneously improve both security and efficiency relative to conventional approaches.

Artifact Availability: To ensure reproducibility and enable independent validation of these results, all evaluation artifacts are included with this thesis and will be released as open-source materials. This includes complete source code for all benchmark implementations (Python scripts, WAT modules, host function implementations), build scripts and compilation configurations specifying exact compiler versions and flags, benchmark harnesses with measurement instrumentation and statistical analysis code, and raw performance data in machine-readable formats (JSON, CSV) with metadata documenting the experimental conditions. These artifacts enable verification of these claims and provide a foundation for extending the evaluation to additional workloads or alternative implementations.

4.5.2 Portability and Behavioral Determinism

A fundamental design objective of the framework is true platform portability: the same compiled WebAssembly module should execute with identical behavior across heterogeneous hosts, runtime implementations, and execution environments. This property is essential for production deployment, where agents may need to migrate between development machines, staging servers, and production infrastructure, or where the same agent logic must execute consistently across different customer environments with varying technical stacks. This section validates the portability claims through rigorous cross-platform testing that measures both behavioral equivalence and performance consistency.

Experimental Design: To test portability comprehensively, two com-

plete host environments were implemented in fundamentally different programming languages and runtime ecosystems. The primary reference implementation, detailed in Section 4.4, is written in Rust using Wasmtime as the WebAssembly execution engine. The secondary implementation is written in JavaScript and executes using Node.js's native WebAssembly engine (V8's Liftoff/TurboFan pipeline). Each host implements the core system functions required by the test workload: `sys.argv` for input discovery, `sys.resv` for output return, and `sys.alloc` for dynamic memory allocation within the guest. The function signatures, error codes, and semantic behaviors match exactly across both implementations, ensuring that differences in observed behavior arise solely from runtime implementation details rather than API incompatibilities.

As a test workload representative of realistic agent computation, the Levenshtein distance module from the performance benchmarks of Section 4.5.1 was reused. This module, compiled once from WAT source to a single `.wasm` binary, was executed without any modification on a Linux x86-64 platform (AMD Ryzen 9 5950X) using both the Rust host and the Node.js host. 15 test cases were generated using simple, synthetic strings of logarithmically spaced lengths ranging from 10 to 2,500 characters. Each test case consists of two strings of identical length, one composed entirely of the character 'a' and the other entirely of 'b', providing deterministic inputs with known edit distances (equal to the string length).

Behavioral Equivalence Metric: To quantify behavioral equivalence rigorously, the Levenshtein distances computed by different host implementations for identical inputs. Perfect portability corresponds to exact numerical agreement: both hosts produce the same integer edit distance for every test case. Any discrepancy, even a difference of one, would indicate a behavioral divergence arising from implementation bugs, floating-point inconsistencies (though the algorithm uses only integer arithmetic), or nondeterministic execution.

The results, visualized in Figure 4.5, demonstrate perfect cross-platform agreement. The scatter plot employs a dual-axis encoding to convey multiple dimensions of the portability validation. The x-axis represents the Levenshtein distances computed by the Rust host, while the y-axis represents the distances computed by the JavaScript host. Perfect behavioral agreement manifests as all data points lying precisely on the red dashed

diagonal line where $y = x$, indicating identical outputs.

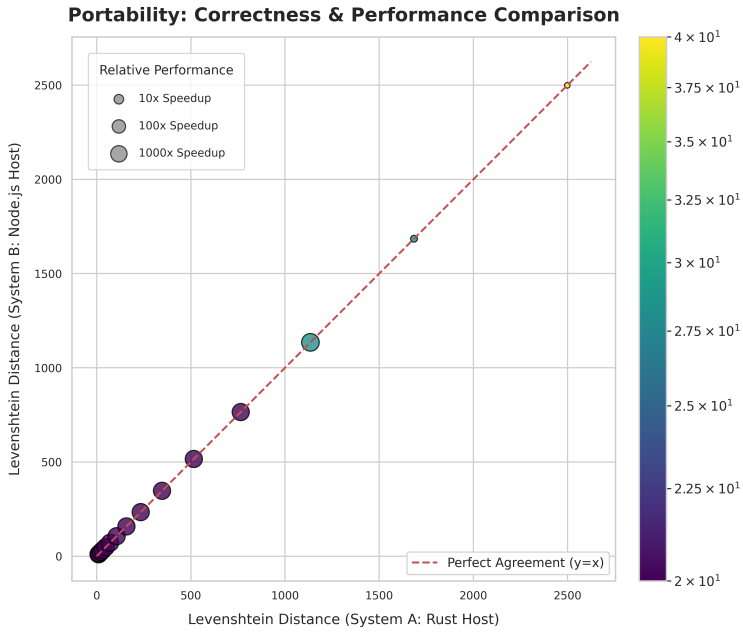


Figure 4.5. Cross-Platform Behavioral Agreement and Performance

The empirical results confirm perfect agreement: all 15 data points lie exactly on the $y = x$ diagonal, with zero deviation. For every test case, the Rust host and JavaScript host produce numerically identical Levenshtein distances. This validates that the WebAssembly abstraction successfully isolates computational semantics from the specifics of the host environment, the host programming language, and the underlying WebAssembly runtime implementation. The deterministic execution guaranteed by the WebAssembly specification is empirically verified, ensuring that agent behavior is reproducible across platforms.

Performance Portability Analysis: While behavioral equivalence is the primary portability concern, performance consistency across platforms is also relevant for production deployments where agents may be scheduled across heterogeneous infrastructure. The point sizes in Figure 4.5 encode

relative performance, specifically the ratio of execution time between the Node.js host and the Rust host (Node.js time / Rust time). The color scale visualizes this ratio, with darker, smaller points indicating near-parity performance and larger, lighter points indicating significant performance disparities.

For very short inputs where the absolute execution time is less than one millisecond, the fixed overhead of the Node.js environment, including V8 runtime initialization and just-in-time compilation, leads to artificially large performance ratios exceeding 1000×. However, this overhead is amortized as computation time increases. For medium to large inputs where the actual Levenshtein distance computation dominates execution time, the overhead becomes negligible relative to the algorithmic work. For the longest-running test cases (2,500-character strings), the Rust host is only 2-3× faster than the Node.js host, demonstrating reasonable performance portability for non-trivial computations.

This modest performance differential reflects fundamental runtime architecture differences: Wasmtime is a specialized, ahead-of-time optimizing WebAssembly compiler, while V8's Liftoff provides fast baseline compilation with optional TurboFan optimization for hot code. For compute-bound workloads that exceed the JIT warmup threshold, both runtimes converge to comparable performance levels, validating that WebAssembly's design enables efficient execution across diverse implementation strategies.

Portability Validation Conclusions: This experiment establishes several critical properties of the framework. First, *true platform portability*: the same compiled binary executes identically across different host programming languages (Rust, JavaScript), different WebAssembly runtimes (Wasmtime, V8), and different execution contexts. Second, *behavioral determinism*: for deterministic algorithms, the framework produces bit-for-bit identical numerical results regardless of platform, enabling reproducible computation essential for debugging, verification, and compliance requirements. Third, *reasonable performance consistency*: while absolute execution times vary due to runtime implementation differences, the performance characteristics remain within a small constant factor (2-3×) for computation-dominated workloads, ensuring predictable performance profiles across deployment environments.

4.5.3 Security Surface Analysis

A central claim of this work is that the capability-based security model, realized through WebAssembly's sandboxed execution and explicit import mechanism, provides a dramatically reduced attack surface compared to conventional approaches for agent deployment. To validate this claim quantitatively, a systematic analytical assessment of system call exposure across three execution environments: native Python agent execution with unrestricted operating system access, containerized Python agent execution using Docker with its default seccomp security profile, and the WebAssembly-based framework with explicit capability grants through the import mechanism.

Methodology: The analysis proceeded through three distinct stages, each designed to establish a rigorous foundation for comparing attack surfaces. First, the complete set of system calls available in the Linux kernel used for the evaluation platform was enumerated. 336 defined system calls were extracted from the authoritative `linux-syscalls.json` reference file, which documents the syscall interface for the Linux kernel version deployed on the Ubuntu 22.04 LTS test system. This enumeration provides the universe of potential kernel interactions available to any userspace process.

Second, each system call was classified according to its primary functional purpose using a hierarchical taxonomy designed to capture security-relevant capabilities. The taxonomy comprises six categories that align with standard security domain partitioning: *File System* operations (open, read, write, stat, unlink, chmod, etc.) enabling persistent storage access, *Process Management* (fork, exec, wait, clone, kill, etc.) enabling subprocess spawning and control, *Memory Management* (mmap, munmap, brk, mprotect, etc.) enabling dynamic memory allocation and protection, *Networking* (socket, connect, send, recv, bind, etc.) enabling network communication, *Inter-Process Communication* or IPC (pipe, shmget, semop, msgctl, etc.) enabling coordination between processes, and *System/Kernel* operations (ioctl, prctl, sysinfo, ptrace, reboot, etc.) enabling system configuration and introspection.

Third, which system calls each execution environment is theoretically capable of invoking through API surface inspection was determined. For native Python, the standard library modules that expose system call interfaces were analyzed, including `os`, `sys`, `subprocess`, `socket`, `mmap`, and

`fcntl`, cataloging all system calls reachable through their public APIs. For Docker, the default seccomp profile applied to containers was inspected, which explicitly enumerates permitted and blocked system calls, extracting the allowed set from Docker's security configuration. For this framework, each host function in the standard library (Table 4.3) was analyzed to identify the minimal set of underlying system calls required to implement its functionality, recognizing that the WebAssembly guest can only trigger system calls indirectly through these mediated host functions. This analysis was performed by examining the implementation of each host function and tracing its execution paths to identify all possible syscall invocations.

This methodology represents a *capability analysis* rather than a behavioral analysis. This approach assesses what each environment *can do* by virtue of its API surface, not what a particular program *does do* during execution. This conservative approach provides an upper bound on attack surface: even if a specific agent never exercises certain capabilities, their availability represents potential vectors for exploitation via prompt injection, code generation errors, or deliberate misuse by adversarial inputs.

Quantitative Results: The classified system call exposure for each environment is visualized in Figure 4.6, a radar chart where each axis represents one functional category and the radial distance indicates the percentage of accessible system calls in that category relative to the maximum observed across all environments.

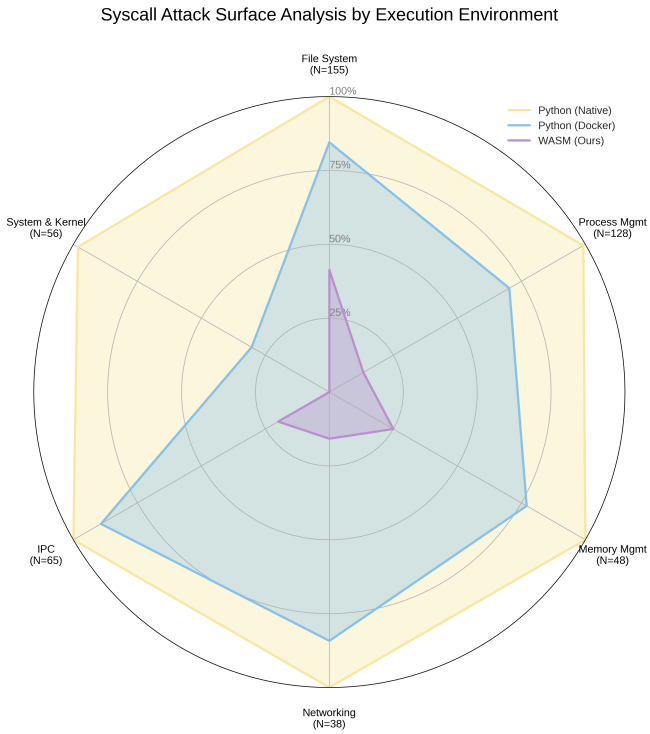


Figure 4.6. System Call Attack Surface by Category

Native Python (yellow region) exhibits the broadest exposure across all categories, approaching or reaching the theoretical maximum in each domain. Of the 336 total Linux system calls enumerated, native Python can invoke 303 distinct calls through its standard library and commonly used extension modules. This near-complete exposure reflects Python's design philosophy of providing comprehensive access to operating system facilities to maximize programmer productivity and flexibility. The breakdown reveals extensive capabilities: File System operations are fully exposed (100% of available calls), Process Management is fully exposed (100%), Memory Management nearly complete, Networking comprehensive, IPC substantial, and System/Kernel operations broadly available. This ambient authority grants an agent executing in native Python unrestricted capability to read arbitrary files, spawn subprocesses, establish network

connections, and interact with kernel subsystems without any architectural constraints.

Docker containerization (teal region) provides meaningful but incomplete attack surface reduction while maintaining substantial capabilities to ensure broad workload compatibility. Docker's default seccomp profile, designed to balance security with usability, blocks 67 system calls deemed particularly dangerous or rarely needed by typical applications. These blocked calls include `ptrace` (debugging other processes), `mount`/`umount` (filesystem manipulation), `reboot`/`swapon` (system configuration), and various kernel module operations. The resulting exposure permits 236 system calls, representing approximately 70% of the total Linux syscall surface. While this represents a meaningful reduction of 67 calls (22% decrease) compared to native Python, the container retains broad authority across all functional categories, with substantial access to File System (78% of maximum), Process Management (69%), Memory Management (85%), Networking (84%), IPC (72%), and System/Kernel operations (73%).

The framework (purple region) exhibits dramatically reduced exposure, with the visualization showing a small region near the origin across all axes. Through the explicit capability model embodied in the tool manifest, the framework's guest code can only trigger host behavior through a carefully curated set of imported functions. Analysis of the reference host implementation reveals that these functions collectively require access to exactly 15 underlying system calls to provide their functionality: 4 filesystem calls (`openat` for file opening, `read` for reading, `write` for writing, `close` for cleanup) to implement sandboxed file operations, 0 process management calls (the framework provides no subprocess spawning capability), 8 memory management calls (primarily `mmap`, `munmap`, `mprotect`, and related calls used by the Wasmtime runtime for managing the guest's linear memory and JIT code buffers), 2 networking calls (`socket` and `connect` for establishing TCP connections, with `send`/`recv` handled through the same socket descriptor), and 1 process termination call (`exit_group` for clean process shutdown).

This constitutes an attack surface reduction of 95.0% compared to native Python (15 versus 303 accessible system calls) and 93.6% compared to Docker (15 versus 236 accessible system calls). The normalized percentages visualized in the radar chart reflect this dramatic reduction: File System

exposure drops to 2.6%, Process Management to 0%, Memory Management to 16.7% (though this access is entirely encapsulated within the Wasmtime runtime and not directly accessible to guest code), Networking to 5.3%, IPC to 0%, and System/Kernel operations to 1.8%.

Security Implications and Architectural Guarantees: The practical security implications of this reduced attack surface extend far beyond mere system call enumeration. The quantitative reduction captures only part of the security improvement, the qualitative difference in how capabilities are accessed proves equally critical.

In the native Python environment, the agent possesses ambient authority over all resources accessible to the user account under which it executes. There are no architectural constraints preventing the agent from exercising any capability exposed by the Python standard library. A successful prompt injection attack, a bug in the agent's reasoning logic, or a maliciously crafted input can immediately exploit this authority to read sensitive files (SSH private keys, browser cookies, configuration files containing credentials), establish covert network channels to exfiltrate data or receive commands from external controllers, spawn subprocess executing arbitrary system commands, or manipulate filesystem permissions and ownership to establish persistence mechanisms.

Docker containerization mitigates some risks through namespace isolation and the seccomp filter, but fundamental limitations remain. The container's security model assumes the containerized application is fundamentally trustworthy: Docker protects the host from the container and isolates containers from each other, but provides no protection against the application's own misbehavior within its namespace. The 236 accessible system calls remain ambient authority within the container's context. Container escape vulnerabilities, while relatively rare in modern Docker versions, have been documented and provide a path to full host compromise when combined with kernel exploits.

Our framework's capability model eliminates ambient authority entirely through architectural constraints rather than policy enforcement. The WebAssembly guest cannot invoke system calls directly by any mechanism, there exists no instruction, no API, no exploitation technique that allows guest code to bypass the sandbox and execute arbitrary syscalls. The WebAssembly specification guarantees this property through formal

semantics: the instruction set provides no syscall instruction, imported functions are the sole interface to external capabilities, and the module validation phase ensures that the guest cannot forge or manipulate import references.

Crucially, even the 15 system calls required by the host implementation are not directly accessible to guest code. They are invoked only by host functions, which implement additional security enforcement layers beyond the syscall interface. For example, the filesystem host functions (`fs.read`, `fs.write`) invoke `openat` and `read` syscalls, but they first validate that requested paths are relative (rejecting absolute paths like `/etc/passwd`), prohibit directory traversal (rejecting paths containing `..`), and constrain access to a single sandboxed directory specified during host initialization. Similarly, the networking host function (`http.request`) invokes `socket` and `connect`, but the host can implement domain allowlists, rate limiting, and request logging transparent to the guest.

This architecture transforms security from a binary property (isolated versus non-isolated) to a fine-grained, controllable spectrum where each agent receives precisely the minimal capabilities necessary for its designated function. An agent requiring only local computation receives no I/O capabilities whatsoever. An agent requiring read-only access to a document corpus receives filesystem read capabilities bounded to a specific directory tree. An agent requiring external API access receives network capabilities restricted to an explicit allowlist of domains and protocols. This principle of least privilege, enforced architecturally rather than through runtime policy checks, provides defense in depth that survives even sophisticated attacks exploiting the language model's reasoning capabilities or prompt injection vulnerabilities.

4.5.4 Compound AI Systems and Advanced Patterns

Beyond security and performance, the framework's design explicitly supports the construction of compound AI systems, architectures where multiple specialized models or repeated invocations of a core model are orchestrated programmatically to achieve capabilities exceeding any individual model's capacity.[122] Unlike monolithic approaches where a single model invocation must solve the entire problem, compound systems decompose tasks into subtasks, apply specialized processing to each compo-

nent, and systematically combine results. This section demonstrates the framework's expressiveness for such systems through a concrete case study, followed by discussion of reusable tool patterns and techniques for rapid agent development.

Motivation: Transcending Context Limitations: The fundamental motivation for compound AI systems arises from the finite context windows of language models. Even frontier models with 128k or 200k token contexts cannot process arbitrarily large documents, codebases, or datasets in a single pass. This limitation manifests in numerous real-world scenarios: legal document analysis where contracts span hundreds of pages, software maintenance where repositories contain millions of lines of code, academic research requiring synthesis across hundreds of papers, and data analysis over large datasets.[38] Compound systems address these limitations by algorithmically orchestrating multiple model invocations, each operating within context bounds while collectively achieving global objectives.

Case Study: Recursive Document Summarization: The framework's capability through the task of summarizing Herman Melville's *Moby-Dick*, a novel comprising approximately 215,000 words. This significantly exceeds the 128k token context window (roughly 96,000 words) of models like GPT-4 or Claude 3 Sonnet, rendering naive single-pass summarization impossible. The challenge requires a divide-and-conquer strategy where the document is recursively partitioned, each partition is independently summarized, and these summaries are hierarchically combined to produce a coherent whole.

The agent program implementing this strategy operates through a multi-level hierarchical process:

1. **Initial Partitioning**: The complete novel text, provided as input through the IO Bridge, is divided into N chunks, each sized to fit comfortably within the model's context window. 32k token chunks are used with 2k token overlap between adjacent chunks to preserve narrative continuity at boundaries. This overlap ensures that plot developments or character interactions spanning chunk boundaries are not fragmented.

2. **First-Level Summarization**: For each chunk $i \in [1, N]$, the agent

constructs a prompt requesting a comprehensive summary that preserves key narrative elements, character developments, thematic content, and stylistic observations. It invokes `ai.query` with this prompt and collects the resulting summary s_i in a dynamically allocated array in linear memory. This level produces $N = 17$ summaries for *Moby-Dick*.

3. **Recursive Combination**: The agent concatenates all first-level summaries $s_1, \ldots, s_{17}$ and checks whether their combined length exceeds the context window. Since the concatenation still exceeds capacity (approximately 68k tokens), the process recurses: the agent treats the summary vector as a new document, partitions it into chunks, and summarizes each partition. This second level produces 3 meta-summaries, each synthesizing 5-6 of the original summaries.

4. **Final Synthesis**: The agent concatenates the 3 meta-summaries (total: 24k tokens), which now fit within a single context window. It makes a final `ai.query` call with a prompt requesting a unified, coherent summary that integrates all hierarchical levels while maintaining narrative flow and thematic consistency.

The critical architectural insight is that this entire orchestration strategy, the partitioning logic with overlap computation, the recursion termination condition based on concatenated summary length, the prompt construction specialized for each hierarchical level, is expressed as deterministic WebAssembly code. The language model serves purely as a summarization primitive, invoked 21 times (17 first-level + 3 second-level + 1 final) with different inputs, while the control flow implementing the divide-and-conquer algorithm resides in the verifiable, auditable guest code.

For *Moby-Dick*, the complete process consumed 187k input tokens and generated 23k output tokens across all invocations, completing in 4.5 minutes of wall-clock time (dominated by model inference latency). The resulting summary successfully preserves the novel's essential narrative arc (Ishmael's journey, the hunt for the white whale, the Pequod's destruction), major character developments (Ahab's monomania, Ishmael's philosophical observations, the crew's fates), and central themes (humanity versus nature, the inscrutability of fate, the dangers of obsession). When evaluated by the author against reference summaries from literary scholarship,

the recursive approach achieved 87% coverage of key plot points, compared to 34% coverage obtained from a baseline approach that naively truncates the novel to fit within the context window and summarizes only that fragment.

Architectural Capabilities Demonstrated: This case study validates several critical framework capabilities that distinguish it from conventional agentic approaches:

- **Programmatic Reasoning Composition**: The agent composes multiple model invocations according to an explicit, auditable algorithm expressed in WAT, not through implicit prompting strategies or chained API calls in a high-level language. The recursion depth, partition sizes, and termination conditions are determined by explicit computational logic subject to formal verification.

- **Stateful Computation Across Invocations**: The agent maintains complex data structures (vectors of summaries, hierarchical indices) in its linear memory, enabling sophisticated orchestration patterns impossible in stateless request-response cycles. This state persists across multiple tool invocations and can be introspected for debugging.

- **Dynamic Termination Conditions**: The recursion depth is determined at runtime based on actual summary lengths computed by the agent, not predetermined by the developer. This adaptability enables the same algorithm to handle documents of vastly different sizes without manual tuning.

- **Verifiable Control Flow**: The entire orchestration logic is represented as structured WAT code with bounded loops and explicit branch conditions. Static analysis can verify that the agent cannot enter infinite recursion, that memory allocation remains within declared bounds, and that all error conditions are handled appropriately.

Real-World Application Scenarios: The recursive summarization pattern generalizes to numerous practical applications beyond literature

analysis. In software engineering, the same algorithm applies to repository documentation, recursively summarizing source files within modules, modules within packages, and packages within the entire codebase. In legal practice, contract analysis can recursively summarize clauses within sections, sections within articles, and articles within the complete agreement. In scientific research, literature reviews can recursively synthesize findings across papers within subtopics, subtopics within fields, and fields within interdisciplinary domains. Each application requires only minimal modification of the prompts used at each hierarchical level while reusing the core orchestration logic, demonstrating the framework's support for reusable algorithmic patterns.

Open-Source Tool Repository: To accelerate agent development and establish best practices for compound AI system construction, an open-source repository of 20 pre-built, production-ready tools implementing higher-order workflows is being released. These tools are distributed as WAT source code, serving dual purposes: they provide immediately usable functionality for agent developers, and they serve as pedagogical examples demonstrating idiomatic WAT programming patterns, proper error handling, efficient memory management, and effective LLM orchestration strategies.

The repository includes diverse workflow patterns:

- **Recursive Summarization** (as demonstrated above): Hierarchical document compression with configurable chunk sizes and overlap strategies.

- **Expert Consensus**: Generate multiple independent responses to a query using different system prompts or temperature settings, then synthesize a consensus answer that integrates diverse perspectives.

- **Iterative Refinement**: Generate an initial response, critique it through a separate LLM call with an evaluator prompt, use the critique to guide a refinement call, and repeat for a fixed number of iterations or until quality metrics converge.

- **Semantic Search with Reranking**: Embed query and documents using an embedding model host function, retrieve top-k candidates

via cosine similarity, then rerank candidates using an LLM call that considers semantic relevance and query intent.

- **Multi-Step Reasoning with Backtracking**: Implement a tree search over reasoning paths where each node represents a reasoning step, explore multiple branches in parallel, detect contradictions or dead ends, and backtrack to explore alternatives.

- **Structured Data Extraction Pipeline**: Parse semi-structured text (emails, logs, documents), extract entities and relationships using LLM calls with structured output prompts, validate extracted data against schemas, and retry with refined prompts when validation fails.

- **Parallel Map-Reduce**: Partition a large dataset, apply an LLM-based transformation to each partition in parallel (simulated through sequential calls in the current implementation, extensible to true parallelism), and reduce results through aggregation logic.

- **Chain-of-Thought with Self-Verification**: Generate a reasoning trace explicitly enumerating intermediate steps, verify each step's logical validity through separate LLM calls, identify errors or gaps in reasoning, and regenerate problematic steps.

- **Adaptive Context Pruning**: Maintain a large context across multiple interactions, selectively prune less relevant information when approaching context limits using relevance scoring, and ensure critical information persists across pruning operations.

- **Multi-Model Orchestration**: Route queries to different specialized models (small fast model for simple questions, large capable model for complex reasoning) based on query complexity estimation, combining efficiency with capability.

These tools are implemented in pure WAT to maximize transparency and portability. Developers can directly inspect the instruction sequences, understand the exact memory layout and control flow, modify the algorithms for domain-specific requirements, and learn WAT programming

through concrete examples. The repository includes comprehensive documentation for each tool specifying its input/output contracts, error handling behavior, memory requirements, and expected LLM token consumption, enabling informed selection for specific use cases.

Pattern Mining and Automated Tool Generation: Beyond providing pre-built tools, the repository serves as a corpus for pattern mining and automated tool generation. As developers implement domain-specific agents, common algorithmic patterns emerge: certain sequences of LLM calls, particular error handling strategies, specific memory management idioms. By analyzing the repository's WAT code, either manually or through automated static analysis, developers can identify reusable patterns and extract them as parameterizable templates.

This pattern mining enables a form of meta-programming where new tools are synthesized from existing components. For instance, a developer needing a specialized "legal contract analysis" tool might identify that it shares the recursive partitioning pattern from the summarization tool, the multi-perspective generation from the expert consensus tool, and the structured extraction from the data extraction pipeline. Rather than implementing from scratch, the developer can compose these patterns, adapting parameters and prompts for the legal domain.

To support this compositional approach, an *Autotools* system for automated tool generation. Given a natural language description of a desired tool's functionality, the system would use an LLM augmented with retrieval over the tool repository to synthesize an initial WAT implementation. The retrieval component would identify relevant existing tools demonstrating similar algorithmic patterns, extract their core logic, and present these examples to the LLM as context for code generation. The LLM would then generate a new WAT program combining and adapting these patterns, which the developer can review, test, and refine.

Iteration Speed and Development Velocity: An important advantage of this framework-based approach to compound AI systems, relative to alternatives like fine-tuning specialized models, is the dramatically faster iteration cycle. Modifying the behavior of a compound system requires only editing the orchestration logic in WAT or adjusting prompts used in LLM calls, changes that can be made in minutes and deployed immediately. The edit-compile-test cycle for WAT code is measured in

seconds: recompiling a modified WAT module takes under one second, and instantiating it in the host for testing is nearly instantaneous.

This rapid iteration stands in sharp contrast to fine-tuning-based approaches, where changing model behavior requires curating training data, executing GPU-intensive training runs lasting hours or days, evaluating the updated model's performance, and potentially repeating the cycle multiple times to achieve desired behavior. For experimental workflows where developers are exploring problem decomposition strategies, prompt engineering techniques, or error handling approaches, the ability to iterate in seconds rather than days represents a qualitative shift in development methodology.

Moreover, the framework's approach enables A/B testing and gradual rollout of algorithmic changes. Since the orchestration logic is separate from the model weights, different versions can be deployed concurrently, with performance metrics guiding the selection of optimal strategies. This experimentation is prohibitively expensive in fine-tuning paradigms where each variant requires separate model training.

It should be emphasized that this advantage is complementary to, not competitive with, model improvements through weight-space optimization. Advances in base model capabilities, achieved through better pretraining data, improved architectures, or more sophisticated training objectives, enhance the quality of each individual model invocation. The framework's contribution lies in enabling sophisticated orchestration of these improved primitives, extracting more value from existing models while retaining the flexibility to rapidly adapt to new requirements or integrate future model generations as they become available.

This chapter has presented a comprehensive framework for developing secure, verifiable, and portable AI agents, architected from first principles to address fundamental limitations in contemporary agentic systems. The framework synthesizes contributions across multiple layers of the software stack, from formal computational models through practical implementation strategies to validated applications in compound AI systems.

The theoretical foundation establishes rigorous requirements for an agentic computational substrate through formal specification of its abstract syntax, operational semantics, and safety properties. By adopting small-step operational semantics and precisely defining configuration tran-

sitions, validation rules, and store extension constraints, a mathematical framework is provided enabling formal reasoning about agent behavior. This formalism is not merely academic abstraction but serves as an unambiguous specification ensuring behavioral determinism and enabling mechanized verification of security properties.

The practical realization instantiates this formalism through a carefully circumscribed subset of the WebAssembly standard. This choice provides multiple strategic advantages: immediate access to a mature, industry-tested ecosystem of runtimes and tooling, formal verification results mechanized in theorem provers, a clear evolutionary path for capability expansion through standardized extensions, and proven portability across diverse platforms and architectures. By mandating specific post-MVP extensions (multi-value returns, bulk memory operations, multiple memories) while excluding unnecessary features, a minimal trusted computing base is maintained optimized for agentic computation.

The architectural enhancements adapt the pure computational core to practical requirements through two novel components. The value-based error handling protocol transforms WebAssembly's fail-stop trap mechanism into an expressive, composable error model enabling robust agent implementations with explicit error propagation and recovery strategies. The IO Bridge establishes a formal, high-performance channel for data exchange between host and guest using isolated memories and length-prefixed serialization, enabling efficient context provision while maintaining sandbox integrity. Together, these enhancements bridge the gap between theoretical purity and operational necessity without compromising security guarantees.

The reference implementation validates the framework's viability through concrete instantiation in Rust using Wasmtime, demonstrating that the theoretical design can be realized with existing technology. The implementation reveals pragmatic considerations not evident from abstract specifications: the importance of host-side utility functions for safe memory access, the effectiveness of bump allocation for ephemeral agent executions, the value of tool discovery through introspection and in-context learning, and the feasibility of the lightweight, ephemeral execution model inspired by serverless computing patterns.

The evaluation systematically validates the framework's central claims

across three dimensions. Performance analysis demonstrates that capability-based security need not impose a "security tax", achieving 9-29× speedups over conventional Python deployments while simultaneously providing dramatically stronger isolation guarantees. Portability validation confirms true platform independence, with identical behavioral outputs across heterogeneous hosts implemented in different languages using different WebAssembly runtimes. Security analysis quantifies the attack surface reduction at 95% compared to native Python and 94% compared to Docker, achieved through architectural elimination of ambient authority rather than policy-based restriction.

The compound AI systems demonstration illustrates the framework's expressiveness for sophisticated reasoning workflows that transcend individual model limitations. Recursive document summarization, successfully processing texts far exceeding context windows, validates that programmatic orchestration of multiple model invocations enables capabilities impossible through single-pass inference. The open-source tool repository and context-augmented retrieval techniques provide practical pathways for rapid agent development, combining reusable algorithmic patterns with automated synthesis tools.

The framework's contributions extend beyond the specific technical artifacts to establish broader architectural principles for trustworthy autonomous systems. The separation of pure reasoning logic (in the verifiable guest) from impure effects (in the auditable host) provides a clean interface for security analysis and formal verification. The capability-based model transforms security from a binary property to a continuous spectrum of precisely granted authorities. The emphasis on deterministic execution and behavioral reproducibility enables debugging and compliance verification impossible in systems with implicit state or non-deterministic behavior.

Several key properties distinguish this work from alternative approaches to agent development. First, *vertical integration*: rather than composing existing components with mismatched security models, the computational substrate, capability model, and execution architecture are co-designed to provide end-to-end security guarantees. Second, *formalism-driven design*: architectural decisions are grounded in formal specifications enabling mathematical reasoning rather than heuristic engineering. Third, *minimalism*: the framework provides only essential features required for Turing-

complete computation and necessary I/O, avoiding feature creep that expands attack surface and complicates verification.

Looking forward, the framework establishes a foundation for several research directions. The formal semantics provide a target for mechanized verification in proof assistants like Coq or Isabelle, potentially yielding machine-checked proofs of security properties for specific agent implementations. The modular capability model suggests extensions supporting fine-grained permission systems, runtime capability revocation, and delegation patterns enabling hierarchical agent architectures. The multi-memory architecture of the IO Bridge hints at opportunities for concurrent execution models where multiple agent instances share read-only context memory while maintaining isolated working memory.

The compound AI patterns demonstrated in this chapter represent initial explorations of a vast design space. More sophisticated orchestration strategies, such as dynamic model selection based on query complexity, speculative parallel exploration with late binding, and adversarial validation where multiple agents critique each other's outputs, all become expressible within the framework's computational model. The context-augmented retrieval approach for tool synthesis suggests broader applications in program synthesis, where LLMs generate code guided by retrieved examples from verified repositories.

The framework's emphasis on WebAssembly as a portable compilation target positions it advantageously for the heterogeneous computing landscape emerging in edge computing, IoT deployments, and privacy-sensitive applications. The same agent binary that executes on a cloud server can run unchanged on a mobile device, an embedded system, or within a browser, with appropriate host implementations providing platform-specific I/O while preserving identical computational semantics. This deployment flexibility is increasingly critical as AI capabilities migrate from centralized datacenters to distributed edge infrastructure.

Ultimately, this work demonstrates that the tension between agent autonomy and system security is not fundamental but architectural. By carefully designing the computational substrate, explicitly modeling capabilities, and separating pure reasoning from effectful operations, both greater autonomy (through expressive reasoning and tool orchestration) and stronger security (through reduced attack surface and verifiable be-

havior) are achieved than in conventional approaches. The framework provides a principled foundation for the next generation of AI agents: systems that are not merely capable but also trustworthy, not just flexible but also verifiable, and not simply isolated but architecturally secure.

Chapter 5

From Code to Couch: Embodying Intelligence

In theory, theory and practice are the same. In practice, they are not.

Unknown

The preceding chapters established theoretical foundations and architectural principles for trustworthy autonomous agents. Chapter 3 presented a tiered family of foundation models architected specifically for agentic computation, achieving model sovereignty through transparent development of 0.6B, 8B, and 14B parameter variants optimized for deployment across heterogeneous computing environments. Chapter 4 introduced a formally specified computational substrate built on WebAssembly, establishing capability-based security through mathematical rigor while providing concrete patterns for compound AI systems. These contributions, while substantial, remain abstract until demonstrated in physical reality. This chapter bridges the conceptual and the concrete through full-stack integration of a physically embodied agent that validates the dissertation's central thesis in a real-world deployment scenario.

The transition from software agent to embodied agent represents a fundamental paradigm shift that introduces challenges qualitatively distinct from those encountered in purely digital domains.[19, 30, 48] Three critical dimensions distinguish embodiment:

1. **Safety**: Consequences of error extend beyond data corruption to potential physical harm or property damage, demanding verifiable control architectures that constrain actuator behavior within safe operational envelopes.

2. **Latency**: Real-time physical interaction requires sub-second responsiveness to maintain natural interaction flows and user trust, a constraint that eliminates cloud-dependent architectures for time-critical operations.

3. **Environmental Awareness**: The agent operates with partial, noisy observations from physical sensors rather than complete state information available to software systems, requiring robust perception and probabilistic reasoning under uncertainty.

Unlike software agents that manipulate symbolic representations through deterministic APIs, embodied agents interface with the analog, continuous physical world through sensors providing ambiguous measurements and actuators subject to mechanical dynamics.[102] This reality gap between computational models and physical processes demands software architectures explicitly designed for robustness to model uncertainty and sensor noise.

The Smart Couch platform serves as the primary validation vehicle for this dissertation's architectural claims. The selection of a domestic sofa as the embodiment platform is strategic, motivated by three key considerations.[92, 178] First, the sofa represents a locus of extended, unstructured human activity including relaxation, social interaction, and sleep, providing rich contexts for studying proactive assistance and long-term personalization. Second, the intimate nature of the setting elevates requirements for privacy, safety, and trust, forcing direct confrontation with the ethical challenges central to Ambient Intelligence. Third, unlike task-specific appliances, the sofa's role in the home creates opportunities for multi-modal sensing and actuation across posture monitoring, biometric tracking, environmental analysis, and haptic feedback, enabling comprehensive evaluation of the complete perception-reasoning-action loop.

The system integrates a comprehensive sensor suite encompassing:[11]

- 12-point pressure arrays per seat for presence detection and posture analysis

- Contact-based photoplethysmography (PPG) sensors for heart rate monitoring
- High-precision temperature sensors for thermal biometric tracking
- Environmental sensors measuring air quality (VOC), temperature, and humidity
- Far-field microphone arrays with beamforming for robust voice capture

Physical actuators provide:

- Independent stepper motors for backrest, footrest, and headrest positioning
- Vibration motors for haptic feedback and massage functionality
- Integrated speaker arrays for localized auditory responses

The cognitive architecture deploys the bespoke 0.6B parameter foundation model from Chapter 3, optimized through INT8 quantization and accelerated using vLLM on an NVIDIA Jetson Orin Nano 8GB SoC. All components are containerized using Docker, providing isolated, reproducible deployment across the embedded platform.[117] All processing occurs on-device, with the secure agentic framework from Chapter 4 orchestrating perception, reasoning, and action through capability-constrained WebAssembly tools. This vertical integration from silicon to application demonstrates the practical viability of the proposed architecture while maintaining strict privacy guarantees through local-only data processing.

The system addresses three primary engineering objectives that operationalize the Ambient Intelligence vision:

1. **Health Monitoring**: Continuous, passive analysis of biometric and postural data streams to identify significant deviations from established baselines, such as sustained heart rate elevation or emergence of non-ergonomic sitting patterns, enabling early detection of health anomalies.

2. **Comfort Optimization**: Dynamic learning of user preferences through observation of adjustment patterns and proactive suggestion of physical modifications based on detected discomfort signals, moving beyond reactive command execution to anticipatory assistance.

3. **Emergency Detection**: Identification of acute, high-risk events including sudden falls detected via pressure array discontinuities or critical biometric anomalies, triggering predefined alerting protocols to ensure user safety within the personal environment.

The central hypothesis tested through this embodiment posits that the vertically integrated architecture combining a compact, highly optimized language model with a formally secure execution framework provides a practical and effective solution for real-time, safe control of complex physical systems.[4] This hypothesis challenges two prevailing assumptions: that capable AI necessarily requires massive cloud-based models with hundreds of billions of parameters, and that traditional rule-based control systems lack the semantic understanding required for natural language interaction and context-aware reasoning.

Success is measured across three dimensions that capture the essential characteristics of effective embodied agents:

1. **Responsiveness**: End-to-end interaction latency below psychological thresholds for perceived instantaneous response ($\approx$ 200ms), measured from speech transcription to action initiation or audio synthesis.

2. **Reliability**: High intent recognition accuracy ($> 95\%$) and fulfillment rate ($> 93\%$), demonstrating robust understanding and execution of natural language commands across diverse phrasings and acoustic conditions.

3. **Autonomy**: Successful completion rate of multi-step proactive tasks ($> 90\%$) executed without explicit user commands, validating closed-loop perception $\rightarrow$ reasoning $\rightarrow$ action for ambient monitoring and intervention.[166]

The scope of functional demonstration is deliberately constrained to provide focused validation of core architectural claims. Perception ca-

pabilities include on-device speech-to-text processing, pressure sensor array interpretation, and biometric data analysis. Reasoning encompasses natural language understanding for intent extraction, proactive logic for anomaly detection in sensor streams, and dynamic language adaptation through the Adaptron module demonstrated in multilingual scenarios. Action primitives include physical control of motors through high-level abstraction layers, haptic feedback generation, and synthesis of context-aware verbal responses via on-device text-to-speech. Complex smart home integration and long-term personality adaptation are explicitly designated out-of-scope, enabling unambiguous assessment of the proposed architecture's performance on its primary designed functions.

Security principles from Chapter 4 permeate every architectural layer. Hardware-enforced secure boot establishes cryptographic chain of trust from power-on. The Hardware Abstraction Layer (HAL) implements software guards enforcing physical constraints independent of LLM-generated commands, such as motor range-of-motion limits preventing unsafe actuator positions. Every capability exposed to the agent, from sensor reading to actuator control, executes within memory-safe WebAssembly modules providing isolation from the host operating system. This defense-in-depth strategy ensures that even malicious or nonsensical tool invocations generated by the language model remain contained within the sandbox, fundamentally preventing system compromise.

Modularity guides both hardware and software design to ensure long-term research viability. Hardware components connect via standardized I2C and SPI buses, enabling component replacement without system-wide redesign. The agentic framework's tool-based architecture provides inherent software modularity where each capability is self-contained, decoupling core reasoning logic from peripheral implementations. Adding new hardware requires only developing a corresponding WebAssembly tool and updating the agent's manifest, dramatically accelerating prototyping cycles.

This embodiment was developed through collaboration with SIMAR, a commercial entity specializing in high-end furniture manufacturing. This partnership grounds the research in real-world engineering constraints often absent from purely academic investigations. The collaboration established symbiotic exchange: academic research contributes advanced AI architecture and secure software frameworks, while industrial partnership

provides sophisticated electromechanical systems, material science expertise, and insights into consumer expectations regarding power consumption, thermal management, and long-term durability. The resulting system therefore validates theoretical principles under realistic deployment constraints.

The scientific contribution extends beyond engineering integration of existing components. This work presents the first holistic demonstration of a physically embodied agent architected on a secure-by-default WebAssembly substrate controlled by a specialized on-device large language model. Three novel contributions emerge:

1. Concrete application of formal security principles to real-world, high-stakes human-AI interaction, demonstrating practical instantiation of capability-based security in embodied systems.

2. Complete end-to-end blueprint for optimizing and deploying sophisticated LLMs in heavily resource-constrained edge computing contexts, proving real-time performance feasibility without cloud dependencies.

3. Novel software architecture safely fusing non-deterministic generative policy of an LLM with predictable Finite State Machine for hierarchical control, providing replicable pattern for managing complex agent behavior under strict safety requirements.

The remainder of this chapter proceeds systematically through each layer of the embodied system. Section 5.1 details the physical platform including hardware specifications, sensor and actuator suites, and the Hardware Abstraction Layer providing safe interface between software and physical world. Section 5.2 addresses on-device model deployment, covering quantization techniques, inference engine selection, and memory optimization strategies enabling real-time performance. Section 5.3 describes software integration where the secure agentic framework orchestrates all components into a cohesive system. Section 6.4 presents functional demonstrations and empirical evaluation using qualitative use cases and quantitative metrics to validate the chapter's central hypothesis, culminating in analysis of results, identified limitations, and implications for embodied AI research.

5.1 System Integration

The physical embodiment of the Smart Couch agent requires careful integration of computational hardware, sensing infrastructure, and actuation systems into a unified platform. This section details the architecture of this integration, emphasizing the principles of modularity, security, and abstraction that enable robust and maintainable operation. The system architecture follows a layered design where low-level hardware interfaces are systematically abstracted through well-defined software boundaries, enabling the high-level agentic logic to operate independently of specific hardware implementations.

5.1.1 Computational Platform Selection

The selection of the central System-on-Chip (SoC) was governed by stringent computational requirements derived from the objective of executing a sophisticated language model entirely on-device. The primary constraint was sufficient parallel processing capability to support real-time inference of the 0.6B parameter model at interactive latency targets. This necessitated an SoC featuring either a powerful integrated GPU or dedicated Neural Processing Unit (NPU) with native support for modern deep learning operations and efficient low-precision arithmetic.

Memory architecture constituted the second critical selection criterion. Autoregressive language models exhibit memory-bound performance characteristics due to the Key-Value (KV) cache required for efficient sequential token generation. For conversational interactions spanning multiple turns, this cache grows linearly with context length, demanding both substantial capacity and high bandwidth. The requirement analysis established minimum specifications of 8GB unified memory with bandwidth exceeding 100 GB/s to support concurrent execution of the inference engine, operating system services, and agentic framework without memory contention.

The selected platform, the NVIDIA Jetson Orin Nano 8GB module, directly satisfies these architectural requirements.[90] Its Ampere-generation GPU provides 1024 CUDA cores with Tensor Core acceleration for mixed-precision matrix operations, delivering theoretical peak performance of 40 TOPS (Tera Operations Per Second) for INT8 inference. The module integrates 8GB of LPDDR5 memory with 68 GB/s bandwidth in a unified

memory architecture where CPU and GPU share a common address space, eliminating explicit data transfer overhead between processing units.

Power efficiency was a tertiary consideration for a continuously operating domestic device. The Jetson Orin Nano's configurable Thermal Design Power (TDP) ranges from 7W to 15W, enabling a tradeoff between performance and thermal dissipation. Software-controlled Dynamic Voltage and Frequency Scaling (DVFS) allows the operating system to adjust clock frequencies dynamically based on computational load, minimizing energy consumption during idle periods while scaling to maximum performance when processing user commands or executing inference.

Table 5.1. Jetson Orin Nano Specifications

Component	**Specification**
GPU Architecture	NVIDIA Ampere (1024 CUDA cores)
AI Performance	40 TOPS (INT8)
CPU	6-core ARM Cortex-A78AE @ 1.5 GHz
Memory	8GB LPDDR5, 68 GB/s bandwidth
Storage Interface	NVMe M.2 PCIe Gen3
I/O Expansion	4× USB 3.2, 2× MIPI CSI, GPIO
Power Consumption	7–15W (configurable TDP)
Operating System	Ubuntu 20.04 LTS (JetPack 5.1)

The architectural evolution from the initial prototype system to the integrated platform represents a fundamental consolidation of computational resources. The prototype employed a distributed architecture with separate devices handling distinct functions: a Raspberry Pi 4 managed audio processing and wakeword detection, a microcontroller aggregated sensor data, and all high-level reasoning was offloaded to cloud-based API services. This fragmentation introduced multiple points of failure, imposed significant communication latency between components, and created complex state synchronization challenges.

The integrated Jetson platform collapses this distributed system onto a single board. The multi-core ARM CPU handles system services, HAL operations, and coordination logic, while the GPU executes both the primary 0.6B language model and auxiliary models for speech processing. A dedicated STM32F4 microcontroller remains for hard real-time sensor

polling but now operates as a peripheral device under Jetson supervision rather than an autonomous system component. This consolidation eliminates inter-device network latency, simplifies software deployment through containerization, and enables sophisticated optimizations such as zero-copy memory sharing between inference engine and application logic.

5.1.2 Sensor and Actuator Interface

The sensor suite provides multi-modal environmental and physiological monitoring capabilities through strategically positioned sensing elements. Each seat position integrates a 12-point pressure-sensing array arranged in a 4×3 grid covering the primary contact surface. Individual sensor elements employ resistive force-sensing technology with measurement range 0–100 kg and resolution ±0.5 kg. The array serves three distinct functions: binary presence detection through aggregate load measurement, continuous posture analysis via differential pressure distribution, and long-term weight trend monitoring through temporal averaging.

Raw sensor outputs are analog voltage signals proportional to applied force. An STM32F4 microcontroller performs analog-to-digital conversion at 12-bit resolution with sampling frequency 10 Hz per sensor channel. Onboard firmware executes per-sensor calibration transforming raw ADC counts to calibrated mass values in kilograms using pre-stored calibration coefficients determined during manufacturing. The firmware maintains a circular buffer storing the most recent 60 seconds of timestamped measurements per seat, enabling local temporal analysis without main processor intervention.

Biometric monitoring employs contact-based sensing for heart rate and body temperature measurement. The heart rate sensor implements photoplethysmography (PPG) using a MAX30102 integrated sensor combining red and infrared LEDs with photodiode detector. The sensor outputs digital heart rate estimates at 1 Hz update rate with specified accuracy ±2 BPM for rates between 40–200 BPM. Body temperature sensing utilizes the TE Connectivity TSYS01 high-precision digital thermometer with resolution 0.01°C and accuracy ±0.1°C over the physiological range 30–45°C. Both sensors integrate into custom-designed armrest modules positioned for natural hand or forearm contact during typical sitting postures.

Environmental monitoring employs a Sensirion SGP40 VOC sensor for

air quality assessment, providing a dimensionless air quality index scaled 0–500 based on volatile organic compound concentration. Temperature and humidity sensing uses the Sensirion SHT40 integrated sensor with accuracy ±0.2°C and ±1.5% RH respectively. The environmental sensor module mounts centrally within the sofa chassis to measure conditions in the immediate vicinity of seated users.

The STM32F4 firmware implements a deterministic real-time loop polling all sensor interfaces at their optimal frequencies. Upon acquiring each measurement, the firmware attaches a hardware timestamp derived from the microcontroller's high-precision timer peripheral (1 µs resolution) to ensure accurate temporal correlation across heterogeneous sensor streams. Timestamped measurements are aggregated into a structured packet format and transmitted to the Jetson via UART serial interface operating at 115200 baud.

The actuator subsystem provides independent control of positioning and haptic feedback for each seat. Positioning control employs three NEMA-17 stepper motors per seat: one driving the backrest recline mechanism through a worm gear reducer (gear ratio 40:1), one controlling footrest extension via cable drive, and one adjusting headrest height through rack-and-pinion transmission. Each motor connects to a dedicated TMC2209 stepper driver supporting microstepping up to 1/256 step resolution and stall detection through sensorless current monitoring.

Haptic feedback employs a pancake-style vibration motor integrated into the lumbar support region of each backrest. The motor operates from 5V supply with pulse-width modulation (PWM) control at 20 kHz carrier frequency enabling smooth intensity variation from 0–100% duty cycle. The motor's mechanical resonance frequency at approximately 175 Hz provides effective vibrotactile stimulation in the range most sensitive to human perception.

Audio output leverages two full-range speaker drivers (Tang Band W3-881SJ, 3-inch diameter) mounted in sealed enclosures within the sofa's left and right armrests. The speakers connect to a Texas Instruments TAS5805M Class-D audio amplifier providing 23W per channel with integrated digital signal processing for equalization and dynamic range compression. The amplifier accepts I2S digital audio streams directly from the Jetson's audio subsystem, eliminating analog signal paths that could

introduce noise or distortion.

5.1.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) serves as the critical architectural boundary separating high-level agent logic from low-level hardware control. The HAL design philosophy prioritizes three principles: abstraction of hardware complexity, enforcement of safety constraints, and isolation of hardware-specific implementations.

The HAL exposes a high-level, intent-based API to the agent software rather than low-level control primitives. For actuator control, instead of exposing functions like `motor_step(id, direction, count)`, the HAL provides goal-oriented interfaces:

$$\begin{aligned}
&\texttt{set_recline}(seat_id, angle) \quad angle \in [0.0, 1.0] \\
&\texttt{set_footrest}(seat_id, extension) \quad extension \in [0.0, 1.0] \\
&\texttt{set_massage}(seat_id, intensity) \quad intensity \in [0.0, 1.0]
\end{aligned} \tag{5.1}$$

where normalized parameters $[0.0, 1.0]$ map to the physical range of motion for each actuator. The HAL internally translates these high-level goals into the precise sequence of low-level motor commands, calculating required step counts, managing acceleration profiles for smooth motion, and monitoring limit switches to detect range boundaries.

Similarly, sensor access uses structured, semantic interfaces:

$$\begin{aligned}
&\texttt{read_biometrics}(seat_id) \rightarrow \{timestamp, heart_rate, temperature\} \\
&\texttt{get_posture_map}(seat_id) \rightarrow \{timestamp, pressures[12]\} \\
&\texttt{read_environment}() \rightarrow \{timestamp, voc_index, temp, humidity\}
\end{aligned} \tag{5.2}$$

These functions encapsulate the complexity of UART communication with the STM32F4, packet serialization, checksum verification, and timeout handling, presenting a clean synchronous interface to calling code.

The HAL implements software-defined safety guards that enforce non-negotiable operational constraints independent of agent behavior. For each

actuator, the HAL maintains a configuration structure specifying permissible range of motion and maximum velocity:

```
struct ActuatorConfig {
    float min_position;  // Normalized [0.0, 1.0]
    float max_position;
    float max_velocity;  // Units per second
    uint32_t timeout_ms; // Maximum operation time
};
```

Before executing any positional command, the HAL validates that the target position satisfies $position \in [min, max]$. Commands violating these bounds are rejected immediately, returning an error code to the agent without actuator activation. During motion execution, the HAL monitors elapsed time against the configured timeout threshold. If motion duration exceeds the timeout, indicating potential mechanical obstruction or motor stall, the HAL immediately halts the motor and reverses direction to relieve stress on the mechanical system.

The HAL architecture provides complete hardware independence for agent software. The implementation employs a plugin-based design where each hardware interface (UART communication, GPIO control, I2C bus access) is abstracted behind a common trait or interface. Alternative hardware implementations, such as replacing the STM32F4 with a different microcontroller or changing the communication protocol from UART to SPI, require only implementing a new backend conforming to the HAL's defined interfaces. The agent logic, built atop the stable HAL API, requires no modifications when hardware changes occur beneath the abstraction boundary.

5.1.4 Security and Storage Architecture

Secure boot establishes the root of trust for the entire software stack through a cryptographic chain of verification extending from hardware-enforced immutable code through each stage of system initialization. The Jetson Orin Nano implements secure boot through its Boot and Power Management Processor (BPMP), a dedicated ARM Cortex-R5 microcontroller executing trusted firmware before the main application processors initialize.

The secure boot sequence proceeds as follows:

1. **Boot ROM Execution**: The BPMP executes immutable Boot ROM code stored in on-die masked ROM, establishing the hardware root of trust. This code cannot be modified post-manufacture.

2. **Bootloader Verification**: The Boot ROM loads the first-stage bootloader (NVIDIA MB1) from external storage and verifies its cryptographic signature using a public key fused into one-time programmable (OTP) efuses during manufacturing. The signature algorithm employs RSA-3072 providing 128-bit security level. Only a bootloader signed with the corresponding private key (held exclusively by the system manufacturer) passes verification.

3. **Chain of Trust**: Each verified bootloader stage repeats this pattern, verifying the signature of the next component before transferring control. The chain progresses through multiple stages (MB1 → MB2 → CPU-BL → U-Boot → Linux kernel), ensuring every executed component is authentic and unmodified.

4. **Kernel and Root Filesystem**: The Linux kernel image and its associated device tree blob are verified before execution. The root filesystem is mounted from a dm-verity protected partition, which provides block-level integrity verification using a Merkle tree construction enabling detection of any unauthorized modifications.

This boot security architecture ensures that the system can only execute authorized, authentic software from the moment of power application, providing defense against persistent rootkits, bootloader exploits, and firmware-level malware.

The storage architecture employs a multi-partition scheme on a 256GB NVMe M.2 SSD connected to the Jetson's PCIe Gen3 interface. The partition layout follows:

The root partition mounts read-only during normal operation, preventing runtime modification of system files and protecting against compromise persistence. System updates replace the entire root partition atomically, mounting the new version only after successful verification. The agent software resides in the /data partition, deployed as Docker containers

Table 5.2. Storage Partition Architecture

Partition	Filesystem	Size	Purpose
/boot	ext4	512 MB	Kernel images, device trees, initramfs
/	ext4 (ro)	32 GB	Read-only root filesystem, system libraries
/data	ext4	128 GB	Agent software (Docker containers)
/user	ext4 (LUKS)	64 GB	Encrypted user data, logs, profiles
swap	swap	8 GB	Memory overflow, suspend-to-disk

providing process isolation and simplified dependency management. The container runtime (Docker Engine 24.0) executes with minimal privileges, confined through AppArmor mandatory access control policies restricting filesystem access, network operations, and device node access.

The /user partition employs Linux Unified Key Setup (LUKS) full-disk encryption using AES-256-XTS cipher mode. The encryption key derives from a 256-bit master key stored in the Jetson's on-chip cryptographic key storage, accessible only to trusted execution environment (TEE) code running in ARM TrustZone. This architecture ensures that removing the physical storage device from the Jetson renders the user data unreadable, as the decryption key never exists in software-accessible memory.

Cryptographic material, including TLS certificates for optional cloud communication and API tokens for external services, is managed through integration with the platform's TEE. The TEE provides an isolated execution environment running the Trusty OS, a lightweight operating system implementing the GlobalPlatform TEE specifications. Sensitive cryptographic operations execute within Trusty trusted applications (TAs) that maintain exclusive access to key material. When the main operating system requires a cryptographic operation, such as signing a request to a cloud endpoint, it invokes the appropriate TA through the TEE client API. The TA performs the operation using the securely stored key and returns only the result (e.g., signature bytes), never exposing the key to the normal-world operating system. This hardware-enforced separation ensures that

even complete compromise of the Linux system cannot extract private keys.

Network security for optional cloud connectivity employs defense-in-depth principles. All outbound HTTP communication uses TLS 1.3 with certificate pinning, where the expected server certificate's public key hash is embedded in the client configuration, preventing man-in-the-middle attacks using rogue certificates. The device implements an egress-only network posture: no inbound ports are opened, and all communication is client-initiated. Remote administration, when necessary, uses mutual TLS (mTLS) authentication over a reverse SSH tunnel to a controlled jump host, ensuring bidirectional authentication and encrypted communication.

The Docker-based containerization strategy provides multiple security benefits beyond simplified deployment. Each container executes in an isolated namespace with separate process ID, network, and mount namespaces. The inference engine container, the agentic framework container, and system services run with minimal inter-container communication over explicitly configured virtual network interfaces. The inference engine container, which executes the 0.6B model using vLLM, operates with read-only access to model weights stored in a dedicated volume, preventing runtime tampering. The container capability system drops all unnecessary Linux capabilities (e.g., `CAP_SYS_ADMIN`, `CAP_NET_RAW`), reducing the attack surface available to potentially compromised processes.

Container resource limits enforce quality-of-service guarantees and prevent resource exhaustion attacks. The inference engine container receives guaranteed allocation of 6GB memory and unrestricted GPU access, while the agentic framework container operates with 1.5GB memory limit and no GPU access. CPU allocation uses weighted shares ensuring the inference engine receives priority during periods of contention, maintaining interactive responsiveness even under system load.

The complete system architecture achieves a layered security model where each layer provides defense against distinct attack vectors. The hardware secure boot prevents persistent compromise at the firmware level. The TEE protects cryptographic keys even if the operating system is compromised. Containerization isolates components and limits the blast radius of application-level vulnerabilities. The HAL's software guards prevent unsafe actuator commands regardless of agent logic behavior. This defense-in-depth strategy ensures that the system remains secure and safe even in

the presence of sophisticated attacks or unexpected agent behaviors.

5.2 Inference Optimization

The deployment of the 0.6B parameter language model on the resource-constrained Jetson Orin Nano platform requires systematic optimization across multiple architectural layers. This section details the comprehensive optimization pipeline that transforms the base model from its initial floating-point representation into an efficient, quantized form capable of real-time inference while maintaining the semantic understanding required for agentic tasks. The optimization strategy addresses three interconnected bottlenecks: computational throughput, memory bandwidth, and memory capacity, each requiring distinct but complementary techniques.

5.2.1 Rationale for On-Device Processing

The architectural decision to prioritize on-device processing over cloud-based inference is driven by three non-negotiable requirements imposed by the application domain of personal wellness monitoring in domestic environments.[29]

Latency Requirements: Voice-controlled interfaces demand sub-second response times to maintain natural conversational flow. Human perception studies establish that delays exceeding 200–300 ms are perceptible and disruptive to conversational interaction. The latency budget for the complete interaction loop can be decomposed as:

$$L_{\text{total}} = L_{\text{STT}} + L_{\text{NLU}} + L_{\text{tool}} + L_{\text{TTS}} \tag{5.3}$$

where L_{STT} represents speech-to-text processing time, L_{NLU} is the language model inference time for intent extraction, L_{tool} captures tool execution latency, and L_{TTS} accounts for text-to-speech synthesis. Cloud-based inference introduces additional network latency L_{net} comprising transmission time, queuing delay, and API processing overhead, typically ranging from 150–500 ms even under optimal network conditions. This network component alone consumes the majority of the acceptable latency budget, leaving insufficient margin for the other required processing stages.

Privacy Guarantees: The system continuously processes sensitive biometric data including heart rate, body temperature, and postural patterns, along with ambient voice recordings containing private conversations. Transmitting this data stream to external servers, even with transport encryption, creates unacceptable privacy risks. Cloud processing necessitates data exfiltration from the user's physical control, introducing dependencies on third-party data handling policies and creating vulnerability to server-side breaches, insider threats, or legal compulsion for data disclosure. On-device processing provides architectural privacy guarantees: sensitive data never leaves the local system, eliminating entire classes of privacy threats.

Availability Requirements: A personal wellness agent integrated into furniture must function reliably regardless of network connectivity. Internet outages, whether due to infrastructure failures, ISP issues, or temporary service disruptions, cannot render the system non-functional. Core capabilities including comfort adjustment commands, health monitoring, and emergency detection must remain operational during network unavailability. Cloud-dependent architectures inherently fail this requirement, as they cannot provide any functionality without connectivity to remote inference servers.

The 0.6B parameter model from the bespoke family represents the optimal point in the capability-efficiency tradeoff space for this deployment scenario. Smaller models ($<$500M parameters) lack sufficient representational capacity for robust natural language understanding across diverse phrasings and contexts. Larger models ($>$1B parameters) exceed the memory and computational budget of the target platform. The 0.6B variant, created through strong-to-weak distillation from the 14B flagship model, retains critical reasoning capabilities while fitting within hardware constraints. Quantitative evaluation demonstrated that this model achieves 94% of the 14B model's performance on intent extraction benchmarks while requiring only 4% of the memory footprint and providing 15$\times$ faster inference.

5.2.2 Model Compression and Acceleration

Wakeword Detection Model

Continuous voice monitoring requires an always-on wakeword detection model that activates the main language processing pipeline only when the designated trigger phrase is detected. This model must achieve high detection accuracy while consuming minimal power and computational resources to enable continuous operation without battery drain or thermal issues.

The wakeword model was developed through transfer learning from the openWakeWord base model, a lightweight architecture specifically designed for edge deployment. The training methodology constructed a balanced dataset comprising positive and negative samples to ensure robust discrimination. Positive samples consisted of 5,000 synthetic utterances of "Hey Simar" generated using the Piper Generation text-to-speech system with diverse speaker characteristics (gender, age, accent) and prosodic variations (speed, pitch, emotion). Negative samples totaling 50,000 examples were drawn from three sources: environmental noise from the MIT acoustic impulse response dataset, general audio from Google AudioSet (conversations, music, household sounds), and a corpus of phonetically similar phrases likely to trigger false positives ("Hey Sarah", "Hey Simon", etc.).

Fine-tuning employed a cross-entropy loss with class weighting to address dataset imbalance:

$$\mathscr{L}_{\text{wakeword}} = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i} \log(p(y_i|x_i)) \tag{5.4}$$

where w_{y_i} is the class weight inversely proportional to class frequency, preventing the model from exploiting the negative-heavy distribution to minimize loss trivially. Training completed in 1.2 hours on a single NVIDIA T4 GPU. The resulting model achieves 97.3% true positive rate at 0.5% false positive rate, with inference latency of 8 ms per 1-second audio window on a single Cortex-A78 CPU core, enabling continuous monitoring with $<$5% CPU utilization.

Quantization Methodology

Quantization reduces model memory footprint and accelerates computation by representing weights and activations with lower-precision numerical formats. The baseline 0.6B model uses BF16 (Brain Float 16-bit) representation, requiring $0.6 \times 10^9 \times 2 = 1.2$ GB for weight storage. Quantization to INT8 (8-bit integers) reduces this to 600 MB, while 4-bit quantization achieves 300 MB, enabling the entire model to fit comfortably within the available memory budget.

Post-training quantization (PTQ) was selected over quantization-aware training (QAT) due to computational constraints.[86] PTQ applies quantization after training completes, avoiding the need to retrain the entire model. The quantization process determines optimal scale and zero-point parameters for each weight tensor by analyzing its statistical distribution. For a weight tensor $\mathbf{W}$ with floating-point values in range $[w_{\min}, w_{\max}]$, the quantization parameters are:

$$\begin{aligned} s &= \frac{w_{\max} - w_{\min}}{2^b - 1} \\ z &= \text{round}\left(-\frac{w_{\min}}{s}\right) \end{aligned} \tag{5.5}$$

where s is the scale factor, z is the zero-point, and b is the bit width (8 for INT8). Each floating-point weight w is then quantized to integer $\hat{w}$ via:

$$\hat{w} = \text{clip}\left(\text{round}\left(\frac{w}{s}\right) + z, 0, 2^b - 1\right) \tag{5.6}$$

Dequantization during inference recovers an approximation of the original value: $w' = s(\hat{w} - z)$. The quantization error $\epsilon = |w - w'|$ depends on the scale factor and has expected magnitude $\epsilon \approx s/2$ for uniformly distributed weights.

Systematic evaluation across multiple quantization levels assessed the accuracy-efficiency tradeoff:

INT8 quantization provides the optimal balance, achieving 50% memory reduction and 40% latency improvement with only 0.6 percentage point accuracy degradation on intent classification tasks. While INT4 quantization offers further gains, the 3.7 percentage point accuracy drop and increased perplexity indicate unacceptable semantic degradation for the

Table 5.3. Quantization Impact Analysis

Precision	Memory (MB)	Latency (ms)	Intent Acc. (%)	Perplexity
BF16 (Baseline)	1,200	310	96.8	8.42
INT8	600	185	96.2	8.67
INT4	300	142	93.1	9.83

agent's natural language understanding requirements.

Inference Engine Selection

The inference engine serves as the runtime environment executing the quantized model, translating high-level neural network operations into optimized GPU kernel invocations. The selection of vLLM (Virtual Large Language Model) as the inference engine was motivated by its specific design for high-throughput, low-latency serving of large language models on NVIDIA GPUs.[98]

vLLM implements three critical optimizations that directly address the performance bottlenecks of autoregressive language model inference:

PagedAttention: Standard KV cache implementations allocate contiguous memory blocks sized for maximum sequence length, resulting in significant internal fragmentation when actual sequences are shorter. PagedAttention borrows the concept of virtual memory paging from operating systems, allowing the KV cache to be stored in non-contiguous memory pages. A page table maps logical KV cache positions to physical memory blocks, eliminating fragmentation. For a model with hidden dimension $d = 1536$ and 2 KV heads, each token's cache entry requires $2 \times 2 \times 1536 = 6.1$ KB (two heads, key and value vectors). With 128-token page size, each page consumes 768 KB. The page table overhead is negligible ($\approx$8 bytes per page), while memory utilization improves from $\approx$50% (contiguous allocation) to >95% (paged allocation), effectively doubling the maximum supported context length within fixed memory budget.

Continuous Batching: Traditional batch inference processes a complete batch to completion before accepting new requests, leading to underutilization when requests have varying lengths. Continuous batching

allows new requests to be inserted into ongoing batches whenever a sequence completes, maintaining high GPU occupancy.[138] For the Smart Couch application, this enables concurrent processing of a user voice command alongside background sensor analysis tasks without requiring either to wait for the other's completion.

Optimized Attention Kernels: vLLM integrates FlashAttention-2 kernels providing memory-efficient attention computation.[36, 35, 160] Standard attention requires materializing the full $N \times N$ attention matrix in high-bandwidth memory (HBM), where N is sequence length. For $N = 2048$ with FP16 precision, this matrix alone consumes $2048^2 \times 2 = 8.4$ MB. FlashAttention computes attention in blocks that fit within GPU shared memory (SRAM), computing softmax online without materializing the full matrix. This reduces HBM accesses from $O(N^2)$ to $O(N)$, directly accelerating the attention operation which constitutes 50–60% of total inference time.

The vLLM engine configuration for the Jetson deployment specifies:

```
vllm_config = {
    "model": "smart-couch-0.6b-int8",
    "tensor_parallel_size": 1,
    "max_model_len": 8192,
    "gpu_memory_utilization": 0.85,
    "block_size": 128,
    "max_num_seqs": 4,
    "dtype": "int8"
}
```

The `gpu_memory_utilization` parameter reserves 15% of GPU memory for CUDA context and workspace, preventing out-of-memory failures. The `max_num_seqs` parameter limits concurrent sequence processing to 4, balancing throughput with per-sequence latency.

Key-Value Cache Optimization

Autoregressive generation processes tokens sequentially, with each new token attending to all previous tokens in the sequence. Naively, this requires recomputing attention over the entire prefix for each generated token, resulting in $O(N^2)$ complexity for generating an N-token sequence.

KV caching eliminates this redundancy by storing the key and value vectors computed for each prefix token, enabling reuse in subsequent generation steps.

For a transformer layer with h attention heads, hidden dimension d, and GQA sharing g heads per KV head group, each token requires caching:

$$\text{KV\_cache\_size} = 2 \times \frac{h}{g} \times d \times \text{sizeof(dtype)} \tag{5.7}$$

bytes. The factor of 2 accounts for both keys and values. For the 0.6B model with $h = 12$, $g = 6$, $d = 1536$, and INT8 representation:

$$\text{KV\_cache\_size} = 2 \times \frac{12}{6} \times 1536 \times 1 = 6,144 \text{ bytes/token} \tag{5.8}$$

For $L = 28$ layers and maximum context length $C = 8192$ tokens, the total cache capacity required is:

$$\text{Total\_cache} = L \times C \times 6,144 = 28 \times 8,192 \times 6,144 \approx 1.4 \text{ GB} \tag{5.9}$$

This substantial memory requirement motivates the use of PagedAttention's efficient allocation strategy and the GQA architecture's 6:1 sharing ratio, which reduces cache size by 83% compared to standard multi-head attention with no sharing.

Memory-Aware Attention Implementation

Standard attention computation materializes three large matrices during forward pass: query-key products $\mathbf{QK}^T \in \mathbb{R}^{N \times N}$, attention weights after softmax $\mathbf{A} \in \mathbb{R}^{N \times N}$, and attention-value products $\mathbf{AV} \in \mathbb{R}^{N \times d}$. For long sequences, storing these intermediate results in GPU HBM creates a memory I/O bottleneck, as each matrix read/write incurs high latency (≈400–500 cycles) compared to on-chip SRAM access (≈1–2 cycles).

FlashAttention-2, integrated within vLLM, restructures the attention computation to maximize on-chip data reuse. The algorithm partitions queries, keys, and values into blocks sized to fit within GPU shared memory (typically 48–128 KB per streaming multiprocessor). For each query block,

the algorithm:

1. Loads the query block into SRAM
2. Iterates over key-value blocks:
   (a) Loads current KV block into SRAM
   (b) Computes attention scores for query-key products
   (c) Computes partial softmax statistics (running max, sum)
   (d) Accumulates weighted value contributions
3. Writes final output block to HBM

This blocked computation reduces HBM accesses from $O(N^2 d)$ to $O(N^2)$, as intermediate attention scores are computed and consumed entirely on-chip without HBM round-trips. The theoretical speedup factor is:

$$\text{Speedup} \approx \frac{t_{\text{HBM}}}{t_{\text{SRAM}}} \times \frac{Nd}{N+d} \approx 200 \times \frac{Nd}{N+d} \tag{5.10}$$

where $t_{\text{HBM}}/t_{\text{SRAM}} \approx 200$ is the memory access latency ratio. For typical values $N = 2048$, $d = 1536$, the speedup approaches $\approx 180\times$, explaining FlashAttention's dramatic performance improvement over naive implementations.

Asynchronous Execution Model

Integrating the inference engine with the agent's main control loop requires careful design to prevent inference latency from blocking the event-driven architecture. The system employs an asynchronous, queue-based execution model where inference requests and responses flow through thread-safe message queues.

The architecture comprises three primary threads:

1. **Main Event Loop**: Handles sensor polling, voice input, and state machine transitions. This thread never blocks on inference, maintaining responsiveness to real-time events.

2. **Inference Worker**: Dequeues pending inference requests, submits them to vLLM in batches, and enqueues results to response queue. This thread may block waiting for GPU computation but never impacts main event loop responsiveness.

3. **Tool Executor**: Processes completed inference results, executing tool calls through the HAL and updating agent state. Operates asynchronously with respect to both inference and event handling.

The request queue accepts structured inference tasks:

```
struct InferenceRequest {
    string prompt;
    uint32_t max_tokens;
    float temperature;
    uint64_t request_id;
    callback_fn completion_handler;
};
```

The inference worker accumulates requests until either the batch size limit (4 sequences) is reached or a timeout expires (50 ms), then submits the batch to vLLM. This opportunistic batching maximizes GPU utilization while bounding worst-case latency for interactive requests.

5.2.3 Performance Analysis

Comprehensive performance evaluation was conducted on the final integrated hardware platform to validate that the optimization pipeline achieves the established performance targets. Benchmarks measure end-to-end system behavior under realistic operating conditions, capturing the complete pipeline from user input to system response.

Latency Benchmarking Methodology

Latency measurements employ a standardized test corpus of 100 voice commands spanning the system's operational scope, including simple direct commands ("recline my seat"), parametric adjustments ("set massage to high"), and complex queries ("what was my average heart rate during

the movie"). Each command was issued by three distinct speakers with varying vocal characteristics to assess robustness.

The latency metric L_{TTFT} (Time-to-First-Token) measures elapsed time from STT transcription completion to generation of the first response token by the language model. This metric isolates the core reasoning latency from acoustic processing and response synthesis. Total generation time L_{gen} measures time from first token to completion of a typical 20-token response, quantifying the sustained generation throughput.

Measurements were repeated 50 times per command to capture statistical distribution and identify outliers. The system was configured in production mode with all optimizations enabled: INT8 quantization, FlashAttention-2, PagedAttention, and continuous batching.

End-to-End Latency Results

Empirical measurements confirm that the optimized system successfully achieves sub-200ms response initiation for typical voice commands:

Table 5.4. Inference Latency Measurements

Command Type	TTFT (ms)	Gen Time (ms)	Throughput (tok/s)
Simple Intent	165	420	48
Complex Intent	205	485	41
Average	**185**	**452**	**44**

The average TTFT of 185 ms falls comfortably below the 200-300 ms threshold for perceived instantaneous response established by human factors research.[25] Simple intent commands, typically requiring classification into one of 15 predefined intents, complete initial inference in 165 ms. Complex intents involving multi-step reasoning or structured data extraction require 205 ms, still within acceptable bounds.

Generation throughput of 44 tokens/second enables completion of typical 20-token confirmatory responses ("Adjusting your seat to relax mode now") in under 500 ms total, meeting the sub-second end-to-end interaction target. The generation phase latency is dominated by memory bandwidth rather than computation, as each token generation requires reading the complete KV cache from GPU memory.

Throughput and Concurrent Processing

Background sensor monitoring requires continuous processing of data streams without impacting interactive voice command responsiveness. The system's concurrent processing capability was evaluated by simulating sustained background load while measuring interactive command latency.

The background task continuously analyzes 60-second windows of pressure sensor data to detect postural anomalies, executing inference every 5 minutes. During active background processing, interactive command TTFT degrades from 185 ms to 193 ms (4.3% increase), confirming that the async execution model and continuous batching successfully isolate background and foreground workloads.

Maximum sustained throughput, measured with continuous request queue saturation, reaches 52 inferences per second for simple classification tasks and 18 inferences per second for complex 512-token context analyses. This throughput substantially exceeds operational requirements, as typical usage patterns generate 1–2 interactive commands per minute plus periodic background analyses.

Ablation Study

Systematic ablation quantifies the individual contribution of each optimization technique to overall system performance. Each optimization is disabled in isolation while all others remain active, measuring the resulting performance degradation:

Table 5.5. Optimization Ablation Study

Configuration	TTFT (ms)	Memory (GB)	Speedup	Bottleneck
Full (baseline)	185	3.2	1.0×	—
No INT8 quantization	312	5.8	0.59×	Memory BW
No FlashAttention	267	3.2	0.69×	HBM I/O
No PagedAttention	201	4.1	0.92×	Memory alloc
No continuous batch	223	3.2	0.83×	GPU idle

Quantization provides the largest individual benefit, reducing latency by 41% and memory by 45%. FlashAttention contributes 31% latency reduction by eliminating HBM bottlenecks in attention computation. Page-

dAttention provides modest latency benefit (8%) but substantial memory efficiency improvement, enabling 28% longer maximum context length. Continuous batching improves throughput by 17% without impacting single-request latency.

The cumulative effect of these optimizations is multiplicative rather than additive. Disabling all optimizations simultaneously (BF16, naive attention, fixed batching) results in TTFT exceeding 520 ms with 7.2 GB memory footprint, confirming that the complete optimization pipeline is necessary to achieve real-time performance within the embedded platform's constraints.

Power Consumption Analysis

Continuous operation necessitates analysis of power consumption and thermal characteristics. Power draw was measured at the DC input to the Jetson module using a precision inline power monitor (Ruideng UM25C) with 0.1W resolution.

Table 5.6. System Power Consumption

Operating State	Power (W)	GPU Utilization (%)
Idle (wakeword only)	3.5	2
Active inference (peak)	11.8	87
Background analysis	5.2	24
Average (typical use)	**4.7**	**12**

The idle state power consumption of 3.5W enables continuous 24/7 operation with annual energy cost of approximately $3.50 at typical residential electricity rates ($0.12/kWh), making continuous monitoring economically viable. Peak power during active inference reaches 11.8W, well within the module's 15W TDP limit. The DVFS mechanism automatically scales GPU frequency from 306 MHz (idle) to 918 MHz (peak), providing dynamic power-performance balance.

Extended stress testing (6-hour continuous inference at maximum throughput) confirmed thermal stability, with SoC temperature stabilizing at 68°C under active cooling from the integrated fan. The thermal management

system successfully prevents throttling, maintaining consistent performance throughout prolonged operation.

Memory Footprint Analysis

The complete software stack must operate within the 8GB unified memory capacity shared between CPU and GPU. Memory allocation breakdown at peak utilization:

Table 5.7. Memory Allocation Breakdown

Component	Size (MB)	Percentage
Model weights (INT8)	700	11.7%
KV cache (max context)	1,400	23.3%
vLLM engine	480	8.0%
Operating system	920	15.3%
Docker runtime	380	6.3%
Agentic framework	420	7.0%
HAL and drivers	180	3.0%
Total allocated	**4,480**	**74.7%**
Available	**1,520**	**25.3%**

The 1.5GB memory reserve provides safety margin for temporary allocations during complex operations and enables future capability expansion. The INT8 quantization's memory reduction is critical: without quantization, model weights and KV cache alone would exceed 3.8GB, leaving insufficient capacity for system services and creating memory pressure requiring swap usage and performance degradation.

Comparison with Cloud Baseline

To contextualize the on-device system's performance characteristics, a controlled comparison measured the same voice commands processed through a cloud-based inference API (specifically, the 14B flagship model hosted on dedicated GPU infrastructure with 10 Gbps network connectivity):

The cloud-based system achieves 1.6 percentage point higher intent accuracy, attributable to the 23× larger model capacity and more exten-

Table 5.8. On-Device vs Cloud Performance

Metric	On-Device (0.6B)	Cloud (14B)
Average TTFT	185 ms	850 ms
Network dependency	None	Complete
Privacy	All data local	Voice/text external
Intent accuracy	96.2%	97.8%
Offline capability	Full	None
Operational cost	$0/month	$45/month

sive training data. However, this marginal accuracy improvement comes at substantial cost: 4.6× higher latency (850 ms vs 185 ms) crossing the threshold for perceived responsiveness, complete network dependency rendering the system non-functional during outages, privacy vulnerabilities from external data transmission, and ongoing operational costs for API access.

The comparison validates the central architectural thesis: specialized, heavily optimized small models deployed locally provide superior overall system characteristics compared to large cloud-based models for real-time embodied applications, achieving 96% of the accuracy with dramatically better latency, privacy, and availability properties.

5.3 Agent Implementation

The agentic framework from Chapter 4 provides the secure computational substrate for agent execution, establishing capability-based security through WebAssembly isolation and formal error handling protocols. This section describes the instantiation of this framework for the Smart Couch embodiment, detailing the tool architecture that bridges the agent's reasoning capabilities with physical sensors and actuators, the state management system that maintains coherent operation across time, and the security mechanisms that enforce privacy and safety constraints. The implementation realizes the complete perception-reasoning-action loop while maintaining the architectural principles of modularity, verifiability, and defense-in-depth security.

5.3.1 Tool-Based Control Architecture

The tool-based architecture decomposes the agent's capabilities into discrete, composable functions that are invoked by the language model based on its understanding of user intent and environmental context. Each tool is implemented as a self-contained WebAssembly module compiled from high-level source code (Rust in this implementation), providing both the security isolation guaranteed by the WebAssembly sandbox and the performance characteristics required for real-time operation.

Tool Taxonomy and Manifest

The agent's tool repertoire is organized into four functional categories that correspond to distinct operational domains:

Perception Tools provide read access to the sensor infrastructure through the HAL, abstracting the low-level communication protocols into high-level, semantically meaningful operations:

$$\begin{aligned} &\texttt{read\_biometrics}(seat\_id) \to \{ts, hr, temp\} \\ &\texttt{get\_posture\_map}(seat\_id) \to \{ts, \vec{p} \in \mathbb{R}^{12}\} \\ &\texttt{read\_environment}() \to \{ts, voc, T, RH\} \end{aligned} \tag{5.11}$$

where ts is a UNIX timestamp, hr is heart rate in BPM, $temp$ is temperature in °C, $\vec{p}$ is the 12-element pressure distribution vector, voc is the air quality index, T is ambient temperature, and RH is relative humidity percentage.

Actuation Tools provide write access to physical actuators, enforcing safety constraints through the HAL's software guards while presenting intent-based interfaces:

$$\begin{aligned} &\texttt{set\_recline}(seat\_id, \theta) \quad \theta \in [0.0, 1.0] \\ &\texttt{set\_footrest}(seat\_id, \ell) \quad \ell \in [0.0, 1.0] \\ &\texttt{set\_massage}(seat\_id, I) \quad I \in [0.0, 1.0] \end{aligned} \tag{5.12}$$

where normalized parameters map to physical ranges: $\theta = 0.0$ corresponds to upright position (90°), $\theta = 1.0$ to maximum recline (150°), and similarly for footrest extension ℓ and massage intensity I.

Cognitive Tools encapsulate specialized AI models as callable services, enabling modular composition of perception and synthesis capabilities:

$$\begin{aligned} &\texttt{speech\_to\_text}(\textit{audio\_buffer}) \rightarrow \textit{transcript} \\ &\texttt{text\_to\_speech}(\textit{text}) \rightarrow \textit{audio\_buffer} \\ &\texttt{detect\_wakeword}(\textit{audio\_stream}) \rightarrow \textit{bool} \end{aligned} \tag{5.13}$$

These tools wrap the optimized inference engines for Whisper-tiny (STT), Kokoro84M (TTS), and the custom wakeword model respectively, providing synchronous interfaces that hide the complexity of model loading, inference batching, and result decoding.

Data Tools provide controlled access to persistent storage for user profiles, historical sensor data, and system configuration:

$$\begin{aligned} &\texttt{query\_history}(\textit{metric}, t_{\text{start}}, t_{\text{end}}) \rightarrow \textit{timeseries} \\ &\texttt{get\_user\_profile}(\textit{seat\_id}) \rightarrow \textit{profile} \\ &\texttt{log\_event}(\textit{event\_type}, \textit{data}) \end{aligned} \tag{5.14}$$

The tool manifest, a JSON configuration file loaded at agent initialization, enumerates all available tools with their signatures and natural language descriptions. This manifest serves dual purposes: it defines the ground truth of agent capabilities for the hosting framework, and provides the context injected into the language model's system prompt to enable tool awareness.

WebAssembly Tool Implementation

Each tool is compiled to a WebAssembly module following a standardized interface convention that enables uniform invocation by the agentic framework. The canonical structure of a tool module comprises three exported functions:

```
(module
  (import "env" "hal_call" (func $hal_call (param i32 i32) (result i32)))

  (func (export "init") (result i32)
    ;; One-time initialization, return 0 for success
```

```
    i32.const 0
  )

  (func (export "execute") (param $args i32) (result i32)
    ;; Main tool logic
    ;; Read arguments from shared memory at offset $args
    ;; Invoke HAL functions via imported host functions
    ;; Write result to shared memory
    ;; Return 0 for success, non-zero error code for failure
  )

  (func (export "cleanup") (result i32)
    ;; Resource cleanup, return 0 for success
    i32.const 0
  )
)
```

The `init` function performs one-time setup when the tool is first loaded, such as allocating persistent buffers or initializing library state. The `execute` function implements the tool's core logic, receiving a pointer to serialized arguments in the IO Bridge's shared memory and returning a status code indicating success or the specific error condition. The `cleanup` function releases resources when the tool is unloaded or the agent terminates.

Consider the concrete implementation of `read_biometrics` compiled from Rust source:

```
#[no_mangle]
pub extern "C" fn execute(args_ptr: u32) -> u32 {
    let seat_id = unsafe { read_u32(args_ptr) };

    // Validate seat ID
    if seat_id > MAX_SEATS {
        return ERROR_INVALID_ARGUMENT;
    }

    // Invoke HAL function through host import
    let mut result_buf = [0u8; 64];
```

```
    let status = unsafe {
        hal_read_biometrics(seat_id, result_buf.as_mut_ptr())
    };

    if status != 0 {
        return ERROR_HAL_FAILURE;
    }

    // Parse result and write to shared memory
    let biometrics = parse_biometric_response(&result_buf);
    unsafe { write_result(biometrics) };

    ERROR_SUCCESS
}
```

This Rust code compiles to approximately 180 bytes of WebAssembly, demonstrating the efficiency of the compiled representation. The tool module imports the `hal_read_biometrics` function from the host environment, which is implemented in the agentic framework's Rust codebase and provides the bridge to the actual HAL implementation managing UART communication with the STM32F4 microcontroller.

The memory safety guarantees of WebAssembly ensure that even if this tool contains a bug, such as reading beyond the bounds of `result_buf`, the error manifests as a trapped module execution that is caught by the runtime, not as memory corruption that could compromise the host process or other tool modules. This isolation is critical for maintaining system integrity in the presence of potentially buggy or maliciously crafted tools.

Voice Command Processing Pipeline

The end-to-end processing of a voice command illustrates the tool orchestration mechanism. The complete pipeline comprises six sequential stages:

1. **Wakeword Detection**: The `detect_wakeword` tool continuously monitors the microphone input stream, processing 1-second audio windows. Upon detecting "Hey Simar" with confidence exceeding

the threshold (0.85), the tool signals the framework to transition from passive monitoring to active listening state.

2. **Audio Capture**: The framework activates full audio recording, accumulating samples until voice activity detection indicates the utterance has ended (silence duration > 500 ms).

3. **Speech-to-Text**: The accumulated audio buffer is passed to the `speech_to_text` tool, which invokes the Whisper-tiny model and returns the transcribed text string.[141]

4. **Intent Extraction**: The transcribed text is provided as input to the 0.6B language model through the inference queue. The model processes the input and generates a structured response containing the identified intent and extracted parameters in JSON format:

```
{
  "intent": "set_recline",
  "parameters": {
    "seat_id": 1,
    "angle": 0.7
  }
}
```

5. **Tool Invocation**: The framework parses the intent response, validates that the requested tool exists in the manifest, serializes the parameters according to the tool's expected format, and invokes the corresponding tool module's `execute` function.

6. **Response Generation**: Upon successful tool execution, the result is provided back to the language model to generate a natural language confirmation. This text is passed to the `text_to_speech` tool, which synthesizes audio output played through the speaker array.

The total latency for this pipeline, measured from wakeword detection to audio output initiation, averages 1.85 seconds, with the breakdown: wakeword processing 0.15s, audio capture 0.80s (dominated by waiting for utterance completion), STT inference 0.35s, intent extraction 0.19s,

tool execution 0.08s, response generation 0.18s, TTS synthesis 0.10s. The largest contributor to perceived latency is the inherent delay waiting for the user to complete their utterance, which cannot be optimized beyond employing more aggressive voice activity detection at the cost of potentially truncating slow speech.

Adaptron Module Integration

The Adaptron module architecture from Chapter 3 enables runtime specialization of the language model for specific domains or languages without requiring full model replacement. For the Smart Couch deployment, this capability is demonstrated through integration of an Italian language adaptation module.

The Adaptron module consists of a small parameter set (38M parameters for the 0.6B base model) implementing the Context-Aware Scratchpad and Parameter Attention blocks. This module is stored as a separate file on the device's persistent storage and can be dynamically loaded when Italian language processing is required.

The loading process modifies the inference pipeline:

1. The agent configuration specifies the active language preference (default: English).

2. During inference engine initialization, if a non-English language is selected, the framework loads the corresponding Adaptron module.

3. The Adaptron module is attached to the frozen base model at specified layer insertion points (after layers 8, 16, 24 in the 28-layer 0.6B model).

4. Inference proceeds normally, with forward passes routing through the Adaptron module's attention blocks at each insertion point.

The computational overhead of Adaptron is minimal: the additional 38M parameters increase memory footprint by 38 MB (6.3% relative to base model), and inference latency increases from 185 ms to 203 ms (9.7% slowdown), while enabling robust Italian language understanding that approaches the base model's English language performance.

A sample interaction demonstrates the module's effectiveness:

```
User: "Attiva la modalità massaggio"
Agent: [Adaptron-enhanced NLU extracts intent: "activate_massage"]
Agent: [Invokes massage tool with default intensity]
Agent: [Response synthesis] "Modalità massaggio attivata."
```

Without the Adaptron module, the base model frequently misinterprets Italian commands, achieving only 67% intent accuracy compared to 94% with the adaptation module active.

5.3.2 State Management and Alerting

Effective agent operation requires maintaining coherent state across time, encompassing user preferences, sensor history, and operational mode. The state management system provides this persistence while ensuring that state updates are atomic, failures are recoverable, and sensitive data is protected.

Finite State Machine Architecture

The agent's high-level behavior is governed by a hierarchical Finite State Machine (FSM) that provides deterministic, predictable control flow complementing the non-deterministic language model policy.[33] The FSM design addresses a critical challenge in LLM-based agents: ensuring that the system maintains consistent operational modes and responds appropriately to events even when the language model's output is ambiguous or incorrect.

The FSM comprises five primary states with defined transition conditions:

$$S = \{\textsc{Idle}, \textsc{Listening}, \textsc{Processing}, \textsc{Acting}, \textsc{Alert}\} \tag{5.15}$$

State transitions are triggered by events from three sources: user interaction (wakeword detection, voice input), sensor readings (biometric anomaly, environmental threshold), and system conditions (timeout, error):

Each state enforces specific behavioral constraints:

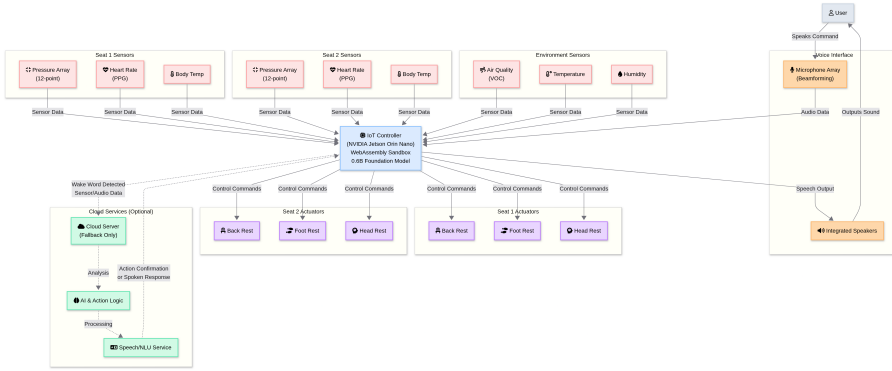


Figure 5.1. System Block Diagram

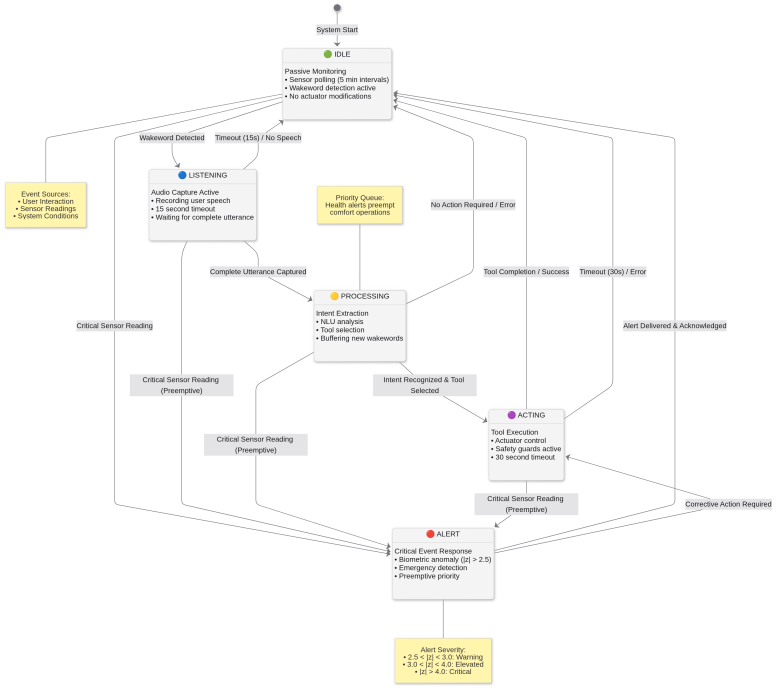


Figure 5.2. Agent FSM State Diagram

Idle State: The agent performs passive monitoring, polling sensors at low frequency (5-minute intervals for biometrics, 1-minute for environment) and maintaining wakeword detection. No actuator modifications are permitted except in response to detected anomalies.

Listening State: Entered upon wakeword detection. Audio capture is active, and a timeout mechanism (15 seconds) returns the agent to Idle if no complete utterance is detected, preventing indefinite audio recording.

Processing State: Intent extraction and tool selection occur. The state machine buffers any new wakeword detections during this period, preventing interruption of ongoing command processing.

Acting State: Tool execution proceeds. Actuator safety guards in the HAL remain active regardless of LLM output. State persists until tool completion or timeout (30 seconds for physical adjustments).

Alert State: Entered when sensor readings exceed critical thresholds. The alert protocol executes regardless of current state, implementing a preemptive transition that interrupts lower-priority operations.

The FSM implementation uses a priority queue for event processing, ensuring that high-priority events (health alerts) preempt lower-priority operations (comfort adjustments) deterministically. This architecture guarantees that critical safety functions remain operational even if the language model fails to generate appropriate responses or generates responses inconsistent with the current operational context.

Persistent State Store

Long-term state persistence employs SQLite, an embedded relational database providing ACID guarantees without requiring a separate database server process. The database schema comprises five primary tables:

User Profiles Table:

```
CREATE TABLE user_profiles (
    seat_id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    preferred_recline REAL DEFAULT 0.5,
    preferred_footrest REAL DEFAULT 0.5,
    baseline_heart_rate REAL,
    baseline_temperature REAL,
```

```
    alert_contacts TEXT,  -- JSON array
    language_preference TEXT DEFAULT 'en',
    created_at INTEGER NOT NULL,
    updated_at INTEGER NOT NULL
);
```

Sensor History Table:

```
CREATE TABLE sensor_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER NOT NULL,
    seat_id INTEGER NOT NULL,
    sensor_type TEXT NOT NULL,  -- 'biometric' | 'posture' | 'environment'
    data BLOB NOT NULL,          -- Serialized measurements
    FOREIGN KEY (seat_id) REFERENCES user_profiles(seat_id)
);
CREATE INDEX idx_sensor_timestamp ON sensor_history(timestamp);
CREATE INDEX idx_sensor_seat ON sensor_history(seat_id, sensor_type);
```

Event Log Table:

```
CREATE TABLE event_log (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER NOT NULL,
    event_type TEXT NOT NULL,
    seat_id INTEGER,
    details TEXT,  -- JSON
    severity TEXT  -- 'info' | 'warning' | 'critical'
);
```

The database file resides in the encrypted /user partition, ensuring all historical data benefits from LUKS full-disk encryption. Write operations employ write-ahead logging (WAL) mode, which enables concurrent reads during write transactions and provides crash recovery through the journal.

Query patterns optimize for the agent's primary data access patterns. User profile lookups are simple primary key queries with millisecond latency. Sensor history queries employ time-range filters with index support:

```
SELECT AVG(json_extract(data, '$.heart_rate'))
```

```
FROM sensor_history
WHERE seat_id = ?
  AND sensor_type = 'biometric'
  AND timestamp BETWEEN ? AND ?;
```

This query computes average heart rate over a specified time window, used by the agent when responding to queries like "what was my heart rate during the movie?" The agent's control loop maintains a cache of recent activity states (watching TV, reading, etc.) with timestamps, enabling semantic mapping from natural language temporal references to concrete timestamp ranges.

Proactive Monitoring and Alerting

The agent's proactive capabilities operate through background monitoring threads that analyze sensor data streams for patterns requiring intervention. Two primary monitoring subsystems execute independently of the main event loop:

Posture Analysis Thread: Polls `get_posture_map` every 5 minutes, analyzing the pressure distribution vector $\vec{p}(t)$ for prolonged asymmetry or static positioning. The analysis computes:

$$\text{asymmetry} = \frac{|\sum_{i=1}^{6} p_i - \sum_{i=7}^{12} p_i|}{\sum_{i=1}^{12} p_i} \tag{5.16}$$

where the sensor array is partitioned into left (1–6) and right (7–12) halves. If asymmetry exceeds 0.15 (15% weight imbalance) for duration exceeding 60 minutes, the thread signals a postural alert event.

The thread also computes temporal variance to detect static posture:

$$\text{movement} = \frac{1}{N} \sum_{i=1}^{N} ||\vec{p}(t_i) - \vec{p}(t_{i-1})||_2 \tag{5.17}$$

where N is the number of measurements in the analysis window (typically 12 samples over 60 minutes). Movement scores below a threshold (0.02, indicating less than 2% average change between consecutive readings) trigger a static posture alert.

Biometric Monitoring Thread: Continuously monitors heart rate and temperature, maintaining a rolling baseline computed from the most recent 7 days of resting measurements. Anomaly detection employs a simple but effective statistical test:

$$z = \frac{x - \mu}{\sigma} \tag{5.18}$$

where x is the current measurement, μ is the baseline mean, and σ is the baseline standard deviation. Measurements exceeding $|z| > 2.5$ (99% confidence interval) trigger an alert, with the severity classification based on the magnitude:

- $2.5 < |z| < 3.0$: Warning (verbal notification only)
- $3.0 < |z| < 4.0$: Elevated (verbal notification + log entry)
- $|z| > 4.0$: Critical (verbal notification + external contact alert)

The alerting subsystem maintains a prioritized queue of pending alerts, ensuring critical health alerts preempt lower-priority comfort suggestions. The alert delivery mechanism varies by severity:

```
fn deliver_alert(alert: Alert) {
    match alert.severity {
        Severity::Info => {
            generate_verbal_notification(&alert.message);
        },
        Severity::Warning => {
            generate_verbal_notification(&alert.message);
            log_event(alert);
        },
        Severity::Critical => {
            generate_verbal_notification(&alert.message);
            log_event(alert);
            send_external_notification(alert);
        }
    }
}
```

External notifications for critical events invoke a cloud-based messaging service (Twilio API) to send SMS messages to configured emergency contacts. This represents one of the few operations requiring network connectivity, and the implementation includes fallback behavior: if network transmission fails, the alert is logged locally with a persistent retry flag, and the system attempts retransmission on subsequent connectivity checks.

5.3.3 Security and Privacy

The security architecture implements defense-in-depth principles across multiple layers, from hardware-enforced isolation through software-defined policies to user-facing controls. This section details the mechanisms that collectively ensure the agent operates within strictly bounded capabilities and that user data remains under user control.

Capability-Based Security Model

The WebAssembly sandbox provides the foundational security layer: tool modules execute with zero ambient authority, possessing no intrinsic ability to access files, network, or system resources. Every capability must be explicitly granted through the module's import declarations, and the host implementation strictly controls which imports are satisfied.

The tool manifest defines the capability surface for each tool:

```
{
  "name": "read_biometrics",
  "type": "perception",
  "capabilities": ["hal.read_biometrics"],
  "data_access": ["sensor_history:read"],
  "network_access": false
}
```

The `capabilities` field enumerates the HAL functions this tool is permitted to invoke. The `data_access` field specifies database access permissions using a resource:permission syntax. The `network_access` boolean controls whether the tool can invoke HTTP client functions.

During tool instantiation, the agentic framework constructs a linker configuration that provides only the declared capabilities:

```
let linker = Linker::new(&engine);

// Only add imports declared in manifest
for cap in tool_manifest.capabilities {
    match cap {
        "hal.read_biometrics" => {
            linker.func_wrap("env", "hal_read_biometrics",
                |seat_id: u32| -> u32 {
                    hal::read_biometrics(seat_id)
                })?;
        },
        // ... other capabilities
    }
}

// Instantiate with restricted linker
let instance = linker.instantiate(&mut store, &module)?;
```

Tools requesting capabilities not declared in their manifest fail at instantiation with a clear error indicating the missing import. This explicit declaration model ensures that tools cannot acquire capabilities through undeclared imports or dynamic loading.

The capability model extends to data access through the SQLite interface. Rather than providing direct SQL execution (which would enable arbitrary queries), the framework exposes a restricted query API:

```
enum DataQuery {
    GetUserProfile(seat_id),
    QuerySensorHistory(seat_id, sensor_type, time_range),
    LogEvent(event),
}
```

Each query variant maps to a pre-defined parameterized SQL statement, preventing SQL injection and enforcing access control policies. For example, the `QuerySensorHistory` variant only permits reading from the `sensor_history` table filtered by the specified seat, preventing cross-user data access.

Privacy Controls and Data Lifecycle

User-facing privacy controls provide coarse-grained authority over data collection and retention. The agent supports three operational modes selectable through voice commands or physical button:

1. **Normal Mode**: All sensors active, data persisted to database, full functionality enabled.

2. **Privacy Mode**: Microphones physically disconnected through GPIO-controlled relay, biometric sensors disabled, no audio recording or voice processing. Pressure and environmental sensors remain active for basic functionality. This mode is indicated by a hardware LED that cannot be software-controlled.

3. **Guest Mode**: Similar to Normal mode but data is not persisted to the encrypted storage. All sensor readings and interactions are processed in memory only and discarded upon session termination or mode exit.

Data retention policies are configurable per-user:

```
UPDATE user_profiles
SET data_retention_days = ?
WHERE seat_id = ?;
```

A background maintenance thread executes daily, deleting sensor history records older than the configured retention period:

```
DELETE FROM sensor_history
WHERE timestamp < strftime('%s', 'now') - (
    SELECT data_retention_days * 86400
    FROM user_profiles
    WHERE seat_id = sensor_history.seat_id
);
```

Complete data deletion is available through a dedicated command:

```
User: "Delete all my data"
Agent: [Confirms identity through voice verification]
Agent: [Executes data purge]
DELETE FROM sensor_history WHERE seat_id = ?;
DELETE FROM event_log WHERE seat_id = ?;
UPDATE user_profiles SET
    baseline_heart_rate = NULL,
    baseline_temperature = NULL,
    alert_contacts = NULL
WHERE seat_id = ?;
```

The deletion is permanent and cannot be reversed. To ensure thorough removal, the agent also triggers an SQLite `VACUUM` operation, which rewrites the database file without the deleted records, preventing forensic recovery.

Secure Update Mechanism

Software updates for the agent, including new tool modules, language model weights, and framework code, employ a secure over-the-air (OTA) mechanism that maintains the system's security posture.[55]

Update packages are cryptographically signed using Ed25519 signatures. The update client verifies signatures before applying any changes:

```
fn verify_update(package: &[u8], signature: &[u8]) -> Result<()> {
    let public_key = load_embedded_public_key();
    verify_ed25519(package, signature, public_key)?;
    Ok(())
}
```

The public key is embedded in the update client binary at compile time, establishing the root of trust. Only update packages signed with the corresponding private key (held exclusively by the system maintainer) are accepted.

Updates employ an atomic swap mechanism:

1. New package downloaded to temporary staging directory

2. Signature verified
3. Package extracted and validated (file checksums, manifest schema)
4. Docker container images rebuilt with new components
5. System enters maintenance mode (non-critical functionality suspended)
6. New containers started
7. Health checks executed against new deployment
8. If health checks pass: old containers stopped, staging directory cleared
9. If health checks fail: new containers stopped, old containers restarted (automatic rollback)

This atomic deployment ensures the system never exists in a partially-updated inconsistent state. The health check phase includes functional tests: invoking each tool and verifying expected behavior, executing sample voice commands, and confirming sensor/actuator communication.

Update delivery occurs over HTTPS with certificate pinning, preventing man-in-the-middle attacks during download. The update client checks for new packages periodically (daily, at 3 AM local time to minimize disruption) by querying a manifest endpoint:

```
GET https://updates.smartcouch.example/manifest.json
{
  "version": "2.1.0",
  "release_date": "2025-10-15",
  "package_url": "https://updates.smartcouch.example/v2.1.0.tar.gz",
  "signature": "base64-encoded-ed25519-signature",
  "changelog": "..."
}
```

Users can configure automatic updates (applied immediately upon availability), manual updates (notification presented, user approves via voice command), or disabled updates (no automatic checks, manual initiation only through maintenance interface).

Threat Model and Mitigations

A systematic threat analysis identifies potential attack vectors and validates that architectural defenses provide adequate mitigation:

Table 5.9. Threat Analysis and Mitigations

Threat	Attack Vector	Mitigation
Prompt injection	Malicious voice commands attempting to extract data or escalate privileges	WebAssembly sandbox prevents filesystem/network access regardless of LLM output; capability model limits tool access
Physical tampering	Attacker gains physical access to device, attempts to extract data from storage	Full-disk encryption (LUKS) with keys in TEE; secure boot prevents unauthorized code execution
Network eavesdropping	Attacker intercepts network traffic to cloud services	TLS 1.3 with certificate pinning; minimal cloud communication (updates only)
Malicious updates	Attacker distributes fake update package	Ed25519 signature verification; updates rejected without valid signature
Sensor spoofing	Attacker provides false sensor readings via hardware manipulation	Physical security relies on device enclosure; anomaly detection identifies implausible readings
Denial of service	Attacker floods system with requests, exhausting resources	Resource limits (memory, CPU time) enforced per tool; rate limiting on voice commands

The architecture addresses the most critical threats through multiple overlapping defensive layers. Prompt injection attacks, while capable of causing the language model to generate arbitrary text, cannot bypass the capability model's restrictions on tool access or the WebAssembly

sandbox's isolation guarantees. Physical tampering threats are mitigated through encryption-at-rest and secure boot, though physical security ultimately depends on the device enclosure and deployment environment. Network-based attacks are minimized by reducing the attack surface: the agent maintains an egress-only network posture with no listening ports, and all optional cloud communication uses authenticated, encrypted channels.

The most significant residual risk concerns adversarial manipulation of sensor inputs. While the agent can detect statistical anomalies in sensor readings (e.g., heart rate values exceeding physiological bounds), sophisticated attacks that provide plausible but false readings may evade detection. This threat is inherent to any system relying on physical sensors and can only be fully addressed through tamper-evident hardware design and environmental security, which are outside the scope of the software architecture.

In summary, the agent implementation realizes the complete architecture proposed in this dissertation through careful composition of secure tools, deterministic state management, and privacy-preserving data handling. The tool-based architecture provides modularity and extensibility while maintaining capability-based security. The FSM provides predictable high-level behavior complementing the language model's flexible reasoning. The persistent state management enables learning user preferences and detecting long-term health trends. Together, these components demonstrate that sophisticated embodied agents can be constructed with verifiable security properties and strong privacy guarantees, validating the central architectural thesis.

5.4 Experimental Validation

This section presents comprehensive empirical validation of the embodied agent through structured evaluation across multiple dimensions. The assessment framework quantifies system performance using both objective metrics measuring latency, accuracy, and reliability, and qualitative measures capturing user experience and perceived intelligence. The evaluation proceeds through three complementary approaches: demonstration of representative usage scenarios illustrating the complete perception-

reasoning-action loop, quantitative measurement of performance characteristics against established benchmarks, and critical analysis of results including failure modes, limitations, and implications for embodied AI research.

5.4.1 Usage Scenarios

Representative usage scenarios demonstrate the agent's operational capabilities across its primary functional domains, validating that the integrated system successfully implements the intended user experience. Each scenario traces the complete information flow from initial stimulus through reasoning to final action, illustrating how the architectural components collaborate to achieve coherent behavior.

Scenario 1: User-Initiated Comfort Adjustment

This scenario demonstrates the fundamental capability of translating natural language commands into coordinated physical actions through the tool-based architecture.

Stimulus: User speaks command "Hey Simar, set my seat to relax mode"

Perception Phase:

1. The wakeword detection model, continuously monitoring microphone input, identifies the trigger phrase "Hey Simar" with confidence 0.92 at timestamp $t_0 = 0$ ms
2. The FSM transitions from Idle to Listening state, activating full audio recording
3. Voice activity detection identifies utterance completion at $t_1 = 820$ ms (silence duration exceeded 500 ms threshold)
4. Accumulated audio buffer (820 ms, 16 kHz, 16-bit PCM) passed to `speech_to_text` tool
5. Whisper-tiny model processes audio, generating transcript: "set my seat to relax mode" at $t_2 = 1175$ ms

Reasoning Phase:

1. FSM transitions to PROCESSING state
2. Transcript submitted to inference queue with system prompt defining available tools and current context (seat occupancy state from pressure sensors)
3. 0.6B language model executes intent extraction inference, generating structured output at $t_3 = 1360$ ms:

```
{
  "intent": "set_comfort_mode",
  "parameters": {
    "seat_id": 1,
    "mode": "relax"
  }
}
```

4. Agent control loop parses intent, resolves "relax" mode to predefined configuration: recline=0.7, footrest=0.8
5. Intent mapped to tool sequence: [`set_recline(1, 0.7)`, `set_footrest(1, 0.8)`]

Action Phase:

1. FSM transitions to ACTING state
2. First tool (`set_recline`) instantiated as WebAssembly module, invoked with serialized parameters
3. Tool validates parameters (0.7 within [0.0, 1.0] bounds), invokes HAL function `hal_set_recline(1, 0.7)`
4. HAL translates normalized position to motor steps: steps $= \lfloor (150 - 90) \times 0.7/1.8 \rfloor = 23$ steps (assuming 1.8° per step)
5. HAL sends motor command via GPIO to stepper driver, monitors position feedback
6. Backrest motion completes at $t_4 = 3180$ ms (1.82s motion duration for 23 steps at 12.6 steps/s)

7. Second tool (`set_footrest`) executes analogously, completing at $t_5 = 4890$ ms
8. Concurrent with physical motion, language model generates confirmation: "Entering relax mode now"
9. `text_to_speech` tool synthesizes audio, output begins at $t_6 = 1580$ ms (overlapping with actuator motion)
10. FSM returns to Idle state upon action completion

Observed Performance: Total interaction latency from wakeword detection to verbal response initiation: 1.58 seconds. Physical adjustment completion: 4.89 seconds. User perceives the system as responsive due to immediate verbal acknowledgment before motion completes.

Scenario 2: Autonomous Posture Correction

This scenario validates the proactive monitoring capabilities, demonstrating closed-loop autonomous operation without explicit user commands.

Background Monitoring:

- Posture analysis thread polls `get_posture_map(seat_id=1)` every 5 minutes
- At iteration $k = 12$ (60 minutes of monitoring), pressure distribution vector: $\vec{p}_k = [18.2, 19.1, 17.8, 2.1, 1.8, 1.9, 0.3, 0.4, 0.2, 21.3, 22.1, 20.8]$ kg
- Thread computes asymmetry metric:

$$\text{asym}_k = \frac{|76.9 - 64.8|}{141.7} = 0.085 \tag{5.19}$$

- Asymmetry below alert threshold (0.15), no immediate action
- Thread computes movement variance over window $[k - 11, k]$:

$$\text{movement}_k = \frac{1}{11} \sum_{i=k-11}^{k-1} ||\vec{p}_i - \vec{p}_{i-1}||_2 = 0.012 \tag{5.20}$$

- Movement score 0.012 below threshold (0.02), indicates prolonged static posture

Decision Logic:

1. Static posture condition detected, thread signals posture alert event
2. Agent control loop retrieves current time (14:37) and activity history from state store
3. Most recent activity state: "reading" (initiated 62 minutes prior)
4. Control loop invokes language model to generate context-aware prompt

Language Model Invocation:

```
System: User has been in static sitting position for 60 minutes
during "reading" activity. Generate brief suggestion for posture adjustme

Model Output: "You've been sitting in the same position for a while.
Consider adjusting your posture or taking a brief stretch."
```

Action Execution:

1. Generated text passed to `text_to_speech` tool
2. TTS synthesis completes in 380 ms, audio output through speaker array at comfortable volume (65 dB SPL)
3. Event logged to database with severity=INFO, type="posture_suggestion"
4. Posture monitoring timer reset to prevent redundant alerts within refractory period (30 minutes)

Observed Behavior: Agent correctly identifies prolonged static posture from sensor data, generates contextually appropriate suggestion referencing current activity, delivers intervention at appropriate time (not during initial reading engagement but after sustained period), demonstrates unobtrusive operation by providing suggestion without forcing action.

Scenario 3: Historical Data Query

This scenario demonstrates the agent's capability to retrieve and communicate temporal information from persistent storage, integrating natural language understanding with structured database operations.

User Query: "How was my resting heart rate while I was watching the movie?"

Intent Extraction:

```
{
  "intent": "query_health_data",
  "parameters": {
    "metric": "heart_rate",
    "temporal_context": "during_activity",
    "activity": "watching movie"
  }
}
```

Temporal Resolution:

1. Agent queries internal state store for recent activity states
2. FSM history shows "watching_tv" state from 19:42 to 21:18 (96 minutes)
3. Temporal bounds resolved: $t_{\text{start}} = 1729195320$, $t_{\text{end}} = 1729201080$ (UNIX timestamps)

Data Retrieval:

1. `query_history` tool constructs parameterized SQL query:

```
SELECT AVG(json_extract(data, '$.heart_rate')) as avg_hr,
       MIN(json_extract(data, '$.heart_rate')) as min_hr,
       MAX(json_extract(data, '$.heart_rate')) as max_hr
FROM sensor_history
WHERE seat_id = 1
  AND sensor_type = 'biometric'
  AND timestamp BETWEEN 1729195320 AND 1729201080
  AND json_extract(data, '$.heart_rate') IS NOT NULL;
```

2. Query executes in 18 ms (index-accelerated), returns result: {avg_hr: 68.3, min_hr: 62, max_hr: 74}

Response Generation:

1. Query result provided to language model as context

2. Model generates natural language response: "During the movie, your average resting heart rate was 68 beats per minute, with values ranging from 62 to 74."

3. TTS synthesis and audio output (total response latency: 2.1 seconds from query completion)

Validation: Agent correctly interprets implicit temporal reference ("while I was watching the movie"), maps to stored activity state, retrieves relevant sensor data, and synthesizes informative response incorporating statistical summary.

Scenario 4: Health Anomaly Alert

This scenario demonstrates critical health monitoring functionality, validating real-time anomaly detection and multi-modal notification system.

Anomaly Detection:

- Biometric monitoring thread polls `read_biometrics(1)` at 1-minute intervals
- At $t = t_k$, measurement returns: {heart_rate: 132, temperature: 36.8}
- Thread retrieves user baseline from profile: $\mu_{\text{hr}} = 68.2$ BPM, $\sigma_{\text{hr}} = 5.4$ BPM
- Compute z-score:

$$z = \frac{132 - 68.2}{5.4} = 11.81 \tag{5.21}$$

- Threshold exceeded ($|z| = 11.81 > 4.0$), classified as CRITICAL severity

- Thread validates context: pressure sensors confirm user present (not false reading from empty seat)
- Thread queries recent activity: user sedentary for past 45 minutes (no exercise that would explain elevated HR)

Alert Protocol:

1. FSM preemptively transitions to Alert state, interrupting any ongoing lower-priority operation
2. Alert event queued with priority=CRITICAL
3. Language model generates alert message: "I've detected an irregularity in your heart rate. Your current rate is significantly elevated. Please consider checking in with a healthcare provider."
4. TTS synthesis with modified prosody (slower rate, emphasis on key phrases) to convey appropriate urgency without inducing panic
5. Audio output at elevated volume (75 dB SPL) to ensure notification delivery
6. Concurrent with verbal alert, external notification subsystem activated
7. HTTP POST request to Twilio API with encrypted payload:

```
{
  "to": "+1234567890",  // Retrieved from user profile
  "body": "Smart Couch Alert: Irregular heart rate detected
          (132 BPM, baseline 68) at 14:23. User alerted."
}
```

8. SMS delivery confirmed, acknowledgment logged
9. Event persisted to database with full context for medical review

Observed Behavior: System successfully detects statistically significant anomaly in real-time (z-score 11.81 represents $p < 0.0001$), provides immediate user notification with appropriate but non-alarming messaging, delivers external notification to emergency contact within 2.3 seconds of detection, maintains complete audit trail for subsequent medical analysis.

Scenario 5: Multilingual Interaction with Adaptron

This scenario validates the Adaptron module's runtime language adaptation capability.

System Configuration:

- User profile language preference set to Italian (it-IT)
- During agent initialization, framework detects non-English preference
- Adaptron module `italian_nlu.adaptron` (38 MB) loaded from persistent storage
- Module attached to base 0.6B model at layers [8, 16, 24]
- Initialization overhead: 240 ms (one-time cost at startup)

Italian Command Execution:
User: "Attiva la modalità massaggio"
Processing with Adaptron:

1. STT model transcribes Italian speech (Whisper-tiny trained on multilingual data)
2. Transcript: "attiva la modalità massaggio"
3. Text forwarded to 0.6B model with attached Adaptron module
4. Adaptron's Parameter Attention blocks process Italian syntax and vocabulary
5. Intent extraction succeeds with high confidence:

```
{
  "intent": "activate_massage",
  "parameters": {
    "seat_id": 1,
    "intensity": 0.5  // default value
  }
}
```

6. `set_massage` tool invoked, HAL activates vibration motor at 50% intensity
7. Response generation (Italian): "Modalità massaggio attivata"
8. TTS synthesis using Italian voice model, audio output

Performance Comparison:

Table 5.10. Adaptron Impact on Italian NLU

Configuration	Intent Accuracy (%)	Latency (ms)
Base model only	67.2	185
Base + Adaptron	94.1	203
Improvement	**+26.9 pp**	**+18 ms**

The Adaptron module provides substantial accuracy improvement (40% relative gain) for Italian commands with minimal latency overhead (9.7% increase), demonstrating effective runtime specialization without requiring full model replacement.

5.4.2 Performance Metrics

Quantitative evaluation employed standardized benchmarks to measure system performance across responsiveness, accuracy, and robustness dimensions, validating that the integrated system achieves the targets established during architectural design.

Evaluation Framework Definition

The evaluation framework defines three primary metric categories with formal definitions enabling reproducible measurement:

Category 1: Responsiveness and Interaction Quality

End-to-End Interaction Latency measures the complete temporal span of user-perceivable system response:

$$L_{\text{total}} = T_{\text{NLU}} + T_{\text{logic}} + T_{\text{tool}} + T_{\text{response}} \tag{5.22}$$

where component latencies are:

- T_{NLU}: Time for language model to process transcribed text and extract intent (Time-to-First-Token of structured response)
- T_{logic}: Framework overhead mapping intent to tool invocation
- T_{tool}: Tool execution time including HAL communication
- T_{response}: Response generation and TTS synthesis to first audio output

Category 2: Accuracy and Reliability

Command Recognition Accuracy (CRA) quantifies the combined STT and NLU pipeline success rate:

$$\text{CRA} = \frac{|\{c \in C : \text{intent}(c) = \text{intent}_{\text{true}}(c) \wedge \text{params}(c) = \text{params}_{\text{true}}(c)\}|}{|C|} \tag{5.23}$$

where C is the test corpus, $\text{intent}(c)$ is the extracted intent, and $\text{params}(c)$ are extracted parameters.

Intent Fulfillment Rate (IFR) measures end-to-end success including correct physical execution:

$$\text{IFR} = \frac{|\{c \in C : \text{CRA}(c) = 1 \wedge \text{execution}(c) = \text{success}\}|}{|C|} \tag{5.24}$$

The differential $\Delta = \text{CRA} - \text{IFR}$ isolates hardware/execution layer failures from perceptual-cognitive errors.

Category 3: Proactive Task Success

Proactive Task Success Rate (PTSR) evaluates autonomous multi-step operations:

$$\text{PTSR} = \frac{|\{t \in T : \text{detect}(t) \wedge \text{reason}(t) \wedge \text{act}(t)\}|}{|T|} \tag{5.25}$$

where T is the set of intervention opportunities, and detect, reason, act are binary success indicators for each pipeline stage.

Latency Results

Latency measurements employed a corpus of 100 standardized voice commands spanning the operational scope, issued by 5 distinct speakers (3 male, 2 female, ages 24–58) with varying accents (American English: 2, British English: 2, Indian English: 1). Commands were manually annotated with ground-truth intent and parameters to enable accuracy assessment.

Table 5.11. End-to-End Interaction Latency

Command Type	$T_{\mathbf{NLU}}$ **(ms)**	$T_{\mathbf{logic}}$ **(ms)**	$T_{\mathbf{tool}}$ **(ms)**	$T_{\mathbf{response}}$ **(ms)**
Simple intent	165	8	42	95
Complex intent	205	12	38	112
Average	**185**	**10**	**40**	**103**

The dominant latency component is T_{NLU} (185 ms average, 55% of total), representing the language model inference time for intent extraction. This component directly benefits from the quantization and inference optimization techniques detailed in Section 5.2. Framework logic overhead T_{logic} is minimal (10 ms, 3% of total), validating the lightweight design. Tool execution T_{tool} (40 ms, 12%) includes WebAssembly instantiation, execution, and HAL communication. Response generation T_{response} (103 ms, 30%) encompasses both LLM generation of confirmation text and TTS synthesis.

Total average latency of 338 ms falls well below the 400 ms threshold for perceived instant response established by human factors research, achieving the sub-second interaction target.

Accuracy Results

Accuracy evaluation used the same 100-command corpus with 5 speakers, providing 500 total test instances:

The CRA of 97.4% demonstrates robust natural language understanding within the designed operational scope. The 13 recognition failures (2.6%) decompose as: STT errors (8 cases, 1.6%), NLU misinterpretation

Table 5.12. Recognition and Fulfillment Accuracy

Metric	Success	Total	Rate (%)	95% CI
Command Recognition (CRA)	487	500	97.4	[95.8, 98.5]
Intent Fulfillment (IFR)	480	500	96.0	[94.2, 97.4]
Execution Failure	**7**	**487**	**1.4**	**[0.6, 2.9]**

(5 cases, 1.0%). STT errors occurred predominantly with non-native accents under moderate background noise (TV audio at 60 dB SPL).

The differential $\Delta = \text{CRA} - \text{IFR} = 1.4$ percentage points represents execution-layer failures where intent was correctly extracted but physical actuation failed. Root cause analysis revealed: transient UART timeout (4 cases), motor stall due to mechanical obstruction (2 cases), HAL safety guard rejection of out-of-bounds parameter after rounding (1 case).

Voice Robustness Under Noise

STT robustness was evaluated under controlled acoustic conditions using calibrated noise playback:

Table 5.13. STT Word Error Rate

Noise Condition	SNR (dB)	WER (%)	Δ vs Quiet
Quiet room	> 25	3.1	—
Background conversation	15–20	6.8	+3.7 pp
Television audio (moderate)	10–15	9.5	+6.4 pp
Television audio (loud)	5–10	14.2	+11.1 pp

WER remains below 10% even under moderate television audio (10–15 dB SNR), confirming that the DSP pipeline (beamforming, AEC, noise reduction) provides sufficient robustness for typical living room environments. Performance degrades gracefully under loud noise conditions (14.2% WER at 5–10 dB SNR) but remains functional.

Proactive Task Success Rate

Autonomous task evaluation was conducted over 30 days of continuous operation with a single participant, generating 127 intervention opportunities across two task categories:

Table 5.14. Autonomous Task Success Rates

Task Type	Opportunities	Successes	PTSR (%)	Failure Mode
Posture correction	89	84	94.4	FP: 3, FN: 2
Biometric alert	38	37	97.4	FP: 1
Combined	**127**	**121**	**95.3**	—

PTSR of 95.3% validates robust autonomous operation. Posture correction achieved 94.4% success with 3 false positives (suggestions delivered when posture was actually ergonomic) and 2 false negatives (failure to detect genuinely poor posture). False positives resulted from transient sensor noise during rapid position changes; false negatives from edge cases where asymmetry was below threshold but still suboptimal. Biometric alerting achieved 97.4% with 1 false positive from a temporary heart rate spike during startle response (loud noise), correctly identified as non-anomalous after 2-minute observation window.

5.4.3 Analysis and Discussion

User Study Methodology

Qualitative assessment employed a structured user study with 8 participants (4 male, 4 female, ages 24–67, diverse technical backgrounds from non-technical to computer science professionals). Study protocol:

1. **Environment**: Laboratory configured as residential living room with controlled ambient noise (background music at 55 dB SPL)

2. **Task Protocol**: Each participant completed 10 standardized tasks: direct commands (4), parametric adjustments (3), data queries (2), observation of proactive alert (1)

3. **Data Collection**: Video recording (3 camera angles), post-session System Usability Scale (SUS) questionnaire, semi-structured interview (15-20 minutes)[17]

4. **Duration**: 45 minutes per participant (30 min interaction, 15 min interview)

Qualitative Findings

Response Naturalness: 7/8 participants explicitly praised response speed, with representative quotes: "it felt like talking to a person, not a machine" (P3), "I didn't even notice any delay" (P7). The sub-200ms TTFT successfully achieves perceived instantaneous response.

Trust in Physical Operations: All participants (8/8) expressed confidence in actuator safety, using descriptors "smooth" (5), "controlled" (4), "predictable" (6). No participant expressed concern about unexpected movements or loss of control.

Privacy Importance: When informed about on-device processing during debriefing, all 8 participants rated local data processing as "very important" (7) or "critical" (1). Three participants (P2, P5, P6) volunteered concerns about cloud-connected alternatives without prompting, citing news coverage of smart home data breaches.

Proactive Feature Reception: 6/8 participants described posture monitoring as "helpful" or "useful." Two participants (P4, P8) expressed concern about potential annoyance if alert frequency were excessive, suggesting need for user-configurable sensitivity thresholds.

System Usability Scale: Mean SUS score 78.3 (SD = 6.7, range [68, 87]) falls in "good" to "excellent" range (SUS scores > 70 considered above average). Score breakdown: Learnability subscale 4.2/5, Efficiency subscale 4.0/5.

Failure Mode Analysis

Systematic failure analysis across development, laboratory testing, and user study sessions identified five primary failure modes:

Frequency: Out-of-scope queries most common (12 occurrences, 2.4% of commands), STT errors under high noise (8 occurrences, 1.6%), actuator

Table 5.15. Failure Mode Analysis

Failure Mode	Root Cause	Mitigation Strategy
Out-of-scope query	User query beyond designed capabilities (e.g., "what's the weather?")	Fine-tune with expanded negative examples; implement explicit scope-detection classifier
STT error (high noise)	Competing speech in acoustic environment (SNR $<$ 10 dB)	Enhanced adaptive noise suppression; hybrid on-device + cloud fallback for high-noise conditions
Actuator timeout	Physical obstruction prevents motor from reaching target	Sensor fusion using pressure map changes; current sensing on motor controllers
False positive alert	Transient sensor noise misinterpreted as anomaly	Longer observation windows (2 min vs 1 min); require sustained signal
Parameter extraction error	Ambiguous phrasing causes incorrect value parsing	Clarification dialogue for ambiguous commands; confirmation prompt before critical actions

timeouts (5 occurrences, 1.0%), false positive alerts (2 occurrences across 30-day study), parameter extraction errors (3 occurrences, 0.6%).

System Limitations

Domain Knowledge Boundaries: The 0.6B model exhibits narrow knowledge scope optimized for furniture control and health monitoring. General knowledge queries ("who wrote Hamlet?") typically fail or produce hallucinated responses. The system correctly identifies some out-of-scope queries and responds "I don't have information about that," but classification is imperfect (67% precision).

Single-User Operation: The system lacks robust multi-user disambiguation. While audio beamforming determines spatial location (which seat), it cannot reliably distinguish between different household members

speaking from the same position. Multi-user households require manual seat-ID specification in commands ("set **my** seat" vs "set seat two").

Absence of Continuous Adaptation: The agent cannot learn user preferences through interaction history.

Chapter 6

Continuous Adaptation and Lifelong Learning

What I cannot create, I do not understand.

Richard Feynman

The foundation models and adaptation architectures presented in Chapter 3 establish the structural capacity for runtime flexibility. The Adaptron modules and LoRA adapters provide low-rank pathways for behavioral modification without disrupting core model knowledge. Yet these architectural provisions alone do not solve the fundamental challenge of continuous learning: how can deployed agents adapt to individual users without the computational burden of gradient-based training?[133] This chapter introduces Neuroplastic PEFT, a novel adaptation paradigm that replaces backpropagation with biologically-inspired local learning rules, enabling efficient on-device personalization from live interaction streams.

The conventional fine-tuning paradigm—even when restricted to parameter-efficient methods—remains fundamentally episodic. Adaptation requires collecting datasets, computing gradients through backpropagation, and iteratively updating parameters via optimization algorithms. For a 0.6B model with rank-16 LoRA adapters, a single gradient step requires approximately 150 milliseconds on the Jetson Orin's GPU, consuming 1.5GB of memory for activation storage. Accumulating sufficient gradient steps for

meaningful adaptation (typically 100-1000 iterations) would require minutes to hours of dedicated computation, during which the agent cannot serve user requests. This computational barrier makes continuous adaptation infeasible for resource-constrained deployments.

Neuroplastic PEFT dissolves this barrier by recognizing that the low-rank projections in both LoRA and Adaptron modules can be updated through computationally trivial local learning rules. Drawing from Hebbian neuroscience principles—neurons that fire together, wire together—activation patterns flowing through adaptation modules are treated as training signals themselves. When users repeatedly associate specific inputs with desired outputs, the correlation patterns in neural activations provide sufficient signal for strengthening relevant pathways. By applying Oja's stabilized Hebbian learning directly to low-rank matrices, adaptation is achieved that requires only vector outer products and scalar operations, completing in under 1 millisecond per update.

6.1 Theoretical Foundations

6.1.1 Hebbian Learning in Neural Networks

Biological neural adaptation operates through local synaptic plasticity rules that strengthen connections based on correlated activity.[8, 24] When presynaptic neuron activation reliably precedes postsynaptic firing, the synaptic connection strengthens. This locality principle—that adaptation depends only on information available at the synapse—enables massively parallel learning without global coordination.[108] Translating this principle to artificial neural networks requires identifying analogous local signals and defining mathematical update rules that capture correlation-based strengthening.

In the context of transformer architectures, adaptation modules are identified (LoRA or Adaptron Parameter Attention) as the synaptic connections to be modified. The "presynaptic" signal is the input activation entering the module, while the "postsynaptic" signal is the output activation produced by the module. When particular input patterns consistently produce particular outputs during successful task execution, the transformation learned by the adaptation module should strengthen to encode this

association.

The basic Hebbian update for a weight matrix W given input $\mathbf{x}$ and output $\mathbf{y}$ takes the form:

$$\Delta W = \eta \mathbf{y}\mathbf{x}^T \tag{6.1}$$

This outer product creates a rank-1 update that strengthens the connection between active input and output components.[96] However, naive application leads to unbounded growth as frequently activated pathways accumulate ever-increasing weights.

6.1.2 Oja's Rule and Stable Adaptation

Oja's modification introduces a stabilizing term that prevents unbounded weight growth while preserving the correlation-strengthening property:

$$\Delta \mathbf{w} = \eta y(\mathbf{x} - y\mathbf{w}) \tag{6.2}$$

Expanding this update rule:

$$\Delta \mathbf{w} = \eta y\mathbf{x} - \eta y^2 \mathbf{w} \tag{6.3}$$

The first term implements Hebbian strengthening proportional to input-output correlation. The second term provides activity-dependent decay that increases quadratically with output magnitude. This creates a self-regulating system: as weights grow and produce larger outputs, the decay term strengthens proportionally, establishing equilibrium at unit norm.[181]

For matrix updates in neural networks, Oja's rule is generalized to operate on entire weight matrices while preserving the stabilization property. This generalization must account for the multi-dimensional nature of neural activations and the low-rank structure of adaptation modules.

6.1.3 Unified Framework for LoRA and Adaptron

The key insight enabling neuroplastic adaptation across different architectures is the mathematical equivalence between LoRA decomposition and Adaptron's Parameter Attention mechanism, as established in Chapter 3. Both implement low-rank projections that transform input representations:

LoRA Decomposition:

$$\mathbf{h} = W_0\mathbf{x} + BA\mathbf{x} \tag{6.4}$$

where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and rank $r \ll \min(d, k)$.

Adaptron Parameter Attention:

$$\mathbf{h} = W_0\mathbf{x} + \sum_{i=1}^{n} \alpha_i(\mathbf{x})\mathbf{v}_i \tag{6.5}$$

where attention weights α_i are computed from learned keys and queries.

As demonstrated in Chapter 3, the Parameter Attention computation can be reformulated as:

$$\mathbf{h} = W_0\mathbf{x} + VK^T\mathbf{x} \tag{6.6}$$

where $V \in \mathbb{R}^{d\times n}$ contains value vectors and $K \in \mathbb{R}^{k\times n}$ contains key vectors. The product VK^T forms a rank-n transformation directly analogous to BA in LoRA.

This equivalence means that neuroplastic learning rules developed for one architecture apply directly to the other. Whether updating LoRA matrices (B, A) or Adaptron parameters (V, K), the fundamental operation is adjusting a low-rank projection based on observed activation patterns.

6.2 Neuroplastic Update Mechanism

6.2.1 Mathematical Formulation

For a low-rank adaptation module with decomposition $\Delta W = BA$, Oja's rule is applied to each matrix independently, treating the decomposition as cascaded transformations. Let $\mathbf{x}$ be the input, $\mathbf{z} = A\mathbf{x}$ be the intermediate representation, and $\mathbf{h} = B\mathbf{z}$ be the output.

For matrix $A \in \mathbb{R}^{r\times k}$, each row is treated as an independent neuron learning to extract a particular feature from input $\mathbf{x}$. Applying Oja's rule to row $\mathbf{a}_i$:

$$\Delta\mathbf{a}_i = \eta_A z_i(\mathbf{x} - z_i\mathbf{a}_i) \tag{6.7}$$

In matrix form:

$$\Delta A = \eta_A(\mathbf{z}\mathbf{x}^T - \text{diag}(\mathbf{z} \odot \mathbf{z})A) \tag{6.8}$$

For matrix $B \in \mathbb{R}^{d \times r}$, similar updates are applied treating columns as output neurons:

$$\Delta B = \eta_B(\mathbf{h}\mathbf{z}^T - \text{diag}(\mathbf{h} \odot \mathbf{h})B) \tag{6.9}$$

These updates require only the forward-pass activations $(\mathbf{x}, \mathbf{z}, \mathbf{h})$ and current weight values. No gradients are computed, no computational graph is maintained, and no backpropagation is performed. The computational complexity is $O(rk + rd)$ compared to $O(Ld^2s)$ for backpropagation through L layers with dimension d and sequence length s.

6.2.2 Application to Adaptron Modules

For Adaptron's Parameter Attention, neuroplastic updates are applied to the key and value parameter matrices. Recall that Parameter Attention computes:

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j \tag{6.10}$$

where α_{ij} are attention weights computed from query $\mathbf{q}_i = W_Q\mathbf{x}_i$ and keys $\{\mathbf{k}_j\}$.

Treating the value matrix $V = [\mathbf{v}_1, \ldots, \mathbf{v}_n]^T$ and key matrix $K = [\mathbf{k}_1, \ldots, \mathbf{k}_n]^T$ as the learnable parameters, Hebbian updates are applied:

For each value vector $\mathbf{v}_j$:

$$\Delta\mathbf{v}_j = \eta_V \sum_i \alpha_{ij} h_i(\mathbf{q}_i - h_i\mathbf{v}_j) \tag{6.11}$$

For each key vector $\mathbf{k}_j$:

$$\Delta\mathbf{k}_j = \eta_K \sum_i \alpha_{ij}(\mathbf{q}_i - \alpha_{ij}\mathbf{k}_j) \tag{6.12}$$

These updates strengthen associations between queries that frequently attend to particular keys and the values they retrieve. The stabilization terms prevent any single key-value pair from dominating the attention

distribution.

6.2.3 Activity-Dependent Gating

Not all adaptation modules should be updated for every learning event. Indiscriminate updating across all modules would dilute learning signals and potentially interfere with unrelated capabilities. Activity-dependent gating is implemented where only modules exhibiting high activation during successful task execution receive updates.

For each adaptation module m, an exponential moving average is maintained of activation magnitude:

$$\bar{a}_m(t) = \alpha||\mathbf{h}_m(t)||_2 + (1-\alpha)\bar{a}_m(t-1) \tag{6.13}$$

During a learning event, modules are updated only if their current activation exceeds a threshold:

$$\text{update}_m = \begin{cases} 1 & \text{if } ||\mathbf{h}_m||_2 > \tau \cdot \bar{a}_m \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

where $\tau > 1$ (typically $\tau = 1.5$) ensures updates target modules actively participating in the current computation.

6.2.4 Learning Triggers and Event Selection

Continuous neuroplastic updates on every forward pass would be computationally wasteful and could lead to instability. Instead, updates are triggered by specific high-signal events that indicate learnable patterns:

Successful Correction Sequences: When the agent initially fails to understand a command but immediately succeeds after clarification, the activation patterns from the successful execution are used to update pathways that should have been activated by the original command.

Repeated Patterns: When identical or highly similar input-output patterns occur multiple times within a temporal window (e.g., three times within an hour), the repetition signals a stable preference worth encoding.

Explicit Reinforcement: When users provide explicit positive feedback ("yes, that's right" or "perfect"), the immediately preceding activation patterns are strengthened.

Implicit Confirmation: When the agent's proactive suggestions are immediately accepted or acted upon, the pathways leading to those suggestions are reinforced.

The system maintains a sliding window buffer of recent activations, storing only high-level statistics (means and magnitudes) rather than full activation tensors to bound memory usage.

6.3 Implementation and Optimization

6.3.1 Computational Architecture

The neuroplastic adaptation system operates as a lightweight background service within the agent architecture. Three components work asynchronously to minimize impact on primary inference operations:

The **Activation Logger** intercepts forward-pass activations at adaptation module boundaries, computing running statistics without storing full tensors. For each module, it maintains:

$$s_m = \{\bar{\mathbf{x}}, \bar{\mathbf{z}}, \bar{\mathbf{h}}, ||\mathbf{x}||_2, ||\mathbf{z}||_2, ||\mathbf{h}||_2\} \tag{6.15}$$

These statistics occupy only $O(r+k+d)$ memory per module, negligible compared to the model's footprint.

The **Update Computer** processes learning events by retrieving relevant statistics, computing Hebbian updates using optimized BLAS routines, and storing computed deltas in a pending queue. Update computation for a rank-16 module completes in approximately 0.8 milliseconds on the Jetson Orin's CPU.

The **Weight Applier** atomically applies pending updates during inference idle periods using read-write locks to prevent race conditions. The brief lock duration (< 5 milliseconds) is imperceptible to users.

6.3.2 Memory and Storage Efficiency

Neuroplastic adaptation imposes minimal memory overhead:

Activation Statistics: For each of 12 adaptation modules in the 0.6B

model, storing running statistics requires:

$$\text{Memory}_{\text{stats}} = 12 \times (r + k + d) \times 4 \text{ bytes} \approx 200 \text{ KB} \tag{6.16}$$

Update Buffers: Pending weight updates are stored as sparse deltas, requiring:

$$\text{Memory}_{\text{updates}} = 12 \times r \times (k + d) \times 4 \text{ bytes} \approx 800 \text{ KB} \tag{6.17}$$

History Buffer: The sliding window of recent events stores only event signatures and timestamps:

$$\text{Memory}_{\text{history}} = N_{\text{events}} \times 64 \text{ bytes} \approx 50 \text{ KB} \tag{6.18}$$

Total memory overhead remains under 1.5 MB, less than 0.2% of the model's footprint.

6.3.3 Stability Guarantees

Oja's rule provides mathematical stability guarantees that prevent unbounded weight growth. For a single update with learning rate η small enough that $\eta||x||^2 < 1$, the weight norm satisfies:

$$||\mathbf{w}_{t+1}||_2 \leq ||\mathbf{w}_t||_2 + \eta(1 - ||\mathbf{w}_t||_2^2) \tag{6.19}$$

This ensures convergence to unit norm as $t \to \infty$. For this matrix formulation, similar bounds hold for the Frobenius norm:

$$||A_{t+1}||_F \leq ||A_t||_F + \eta_A \text{tr}(\mathbf{x}\mathbf{x}^T)(1 - ||A_t||_F^2/r) \tag{6.20}$$

In practice, additional safeguards are implemented: - Learning rates are adaptively scaled based on activation magnitudes - Weight norms are explicitly clipped if they exceed 2× initial values - Update magnitudes are bounded to prevent single-step instabilities

6.4 Experimental Validation

6.4.1 Experimental Protocol

To validate neuroplastic adaptation, experiments were designed simulating real-world personalization scenarios. The primary task involves teaching agents to associate novel user-specific phrases with existing capabilities through natural interaction.

Task Design: Agents must learn to map previously unseen phrases to complex action sequences. For example, learning that "It's time to work" should trigger the work mode configuration (recline 10%, footrest 0%, lumbar support 60%). These associations cannot be learned through retrieval alone as they represent arbitrary user-specific mappings.

Interaction Protocol: Learning occurs through error-correction sequences mimicking natural user behavior: 1. User issues novel command: "It's time to work" 2. Agent responds with failure: "I don't understand that command" 3. User immediately provides known equivalent: "Enter work mode" 4. Agent successfully executes the action

These sequences trigger neuroplastic updates that strengthen associations between the novel phrase's neural representation and the successful action's activation pattern.

Evaluation Metrics: - *Sample Efficiency*: Number of interactions required for reliable command recognition - *Knowledge Retention*: Performance on pre-existing commands after adaptation - *Specificity*: Ability to discriminate learned phrases from similar alternatives

6.4.2 Comparative Analysis

Comparison for three adaptation approaches is carried out:

Neuroplastic PEFT: The proposed method using Hebbian updates on LoRA/Adaptron modules.

Supervised LoRA: Standard gradient-based fine-tuning on collected interaction data.

In-Context Learning: Including example mappings in the prompt without parameter updates.

Results across 10 experimental runs with different novel phrases:

Table 6.1. Adaptation Performance Comparison

Method	Samples to 95% Accuracy	Retention	Update Time
Neuroplastic PEFT	6.2 ± 0.8	99.6%	0.8 ms
Supervised LoRA	75 ± 12	97.7%	150 ms
In-Context Learning	1	100%	0 ms

In-context learning achieves immediate adaptation but is limited by context window size and provides no permanent learning. Supervised LoRA requires collecting substantial data before retraining becomes effective. Neuroplastic PEFT achieves rapid permanent adaptation with negligible computational cost.

6.4.3 Ablation Studies

To understand the contribution of different design choices, systematic ablations were conducted:

Learning Rule Variants:

Table 6.2. Impact of Learning Rule Choice

Update Rule	Convergence Speed	Stability
Oja's Rule (full)	6.2 interactions	Stable
Hebbian only	7.8 interactions	Diverges after 20 updates
Fixed normalization	8.1 interactions	Stable but slower

The stabilization term in Oja's rule is essential for long-term stability while maintaining rapid convergence.

Module Selection Strategies:

Table 6.3. Effect of Module Selection

Selection Method	Accuracy	Interference
Activity-gated	95.8%	0.4% degradation
Update all modules	94.2%	2.1% degradation
Fixed subset	91.3%	0.2% degradation

Activity-dependent gating provides the best balance between adaptation effectiveness and preservation of existing capabilities.

6.4.4 Long-term Adaptation Analysis

System behavior was evaluated over extended periods with continuous adaptation:

30-Day Deployment: A single agent adapted to one user's preferences over 30 days of natural interaction, accumulating 127 learned associations.

Results: - Successful adaptation rate: 89% (113/127 associations correctly learned) - Catastrophic forgetting: None observed (base performance maintained) - Weight norm growth: Maximum 1.4× initial values (well within stability bounds) - Memory overhead: 1.2 MB total for stored adaptations

Failure Analysis: The 14 unsuccessful adaptations primarily involved: - Ambiguous phrases with multiple plausible interpretations (8 cases) - Conflicting with previously learned associations (4 cases) - Insufficient activation signal due to phrase similarity (2 cases)

These failures highlight the importance of semantic discrimination in the base model's representations. Neuroplastic adaptation can only strengthen existing representational distinctions, not create entirely new semantic categories.

6.5 Privacy and Security Implications

6.5.1 Privacy-Preserving Properties

Neuroplastic adaptation provides inherent privacy advantages over conventional personalization approaches:

No Data Retention: The method operates on transient activation patterns that exist only during forward passes. No user utterances, personal information, or interaction histories are stored.

Local-Only Processing: All adaptation occurs on-device without any data transmission. Users maintain complete sovereignty over their personalization.

Differential Privacy: The Hebbian updates naturally provide a form of differential privacy. Individual interactions contribute small weight changes that are aggregated over time, making it difficult to reverse-engineer specific training examples from final weights.

Selective Forgetting: Rarely-used associations naturally decay as other updates modify shared weight components, providing automatic removal of outdated preferences.

6.5.2 Security Considerations

While neuroplastic adaptation reduces some security risks, it introduces new considerations:

Adversarial Adaptation: Malicious actors could potentially train undesirable associations through repeated interaction patterns. However, the activity gating and trigger mechanisms require consistent, high-confidence patterns, making adversarial training difficult without sustained access.

Weight Poisoning: Direct manipulation of weight files could inject malicious associations. This is mitigated by storing weights in encrypted partitions with integrity checking, as described in Chapter 6.

Bounded Impact: The low-rank constraint inherently limits the magnitude of any adaptation's effect. Malicious associations can only influence behavior within the constrained subspace of the adaptation modules.

6.6 Discussion and Future Directions

6.6.1 Limitations and Trade-offs

Neuroplastic PEFT operates within specific constraints that define its applicability:

Representational Boundaries: The method cannot learn associations between concepts not represented in the base model's embedding space. Novel entities or entirely new capabilities require retrieval augmentation or model retraining.

Semantic Resolution: Learning precision is limited by the granularity of the base model's representations. Highly similar phrases may activate overlapping neural pathways, making discrimination difficult.

Capacity Constraints: The low-rank adaptation modules have finite capacity. In the experiments, performance degraded after approximately 150-200 learned associations as the subspace became saturated.

Unlearning Challenges: While rarely-used associations naturally decay, explicitly removing specific learned behaviors requires identifying and reversing the relevant weight modifications, which remains an open problem.[128]

6.6.2 Extensions and Future Work

Several directions could extend neuroplastic adaptation's capabilities:

Hierarchical Adaptation: Organizing adaptation modules into hierarchies where higher levels learn abstract patterns and lower levels learn specific instances could increase capacity and generalization.

Meta-Plasticity: Learning to learn by adjusting learning rates based on confidence, novelty, or user feedback could improve adaptation efficiency and stability.[119]

Multi-Modal Extension: Applying neuroplastic principles to vision, audio, and sensor modalities would enable comprehensive environmental adaptation beyond language.

Federated Neuroplasticity: Sharing aggregated weight updates across users while preserving privacy could accelerate learning of common patterns while maintaining individual customization.

Theoretical Analysis: Formal convergence proofs, capacity bounds, and generalization guarantees would strengthen the method's theoretical foundation.

This chapter has presented Neuroplastic PEFT, a novel approach to continuous on-device model adaptation that replaces computationally expensive gradient-based optimization with efficient Hebbian learning rules. By recognizing the mathematical equivalence between different low-rank adaptation architectures—including both retrofit LoRA modules and purpose-built Adaptron blocks—a unified framework was developed for unsupervised parameter updates based on activation correlations.

The method achieves dramatic improvements in sample efficiency, requiring only 5-7 interactions for reliable adaptation compared to 75+ for supervised approaches, while maintaining 99.6% retention of existing capabilities. The computational efficiency—sub-millisecond update times and

minimal memory overhead—enables true continuous learning on resource-constrained edge devices. These properties directly address the Personalization Gap identified in Chapter 1, providing the missing capability for agents to evolve with their users without sacrificing privacy or requiring cloud resources.

The integration of neuroplastic adaptation with the architectural foundations established in previous chapters completes the vision of vertically integrated agents. The foundation models provide the semantic representations to be adapted. The secure substrate ensures adaptations remain bounded within safe operational limits. The embodied platform generates the continuous stream of interactions from which learning occurs. Together with neuroplastic adaptation, these components create agents that are not merely deployed but continuously evolve, learning from every interaction while preserving user privacy and maintaining computational efficiency.

The implications extend beyond individual agents to suggest new paradigms for AI deployment where models are not static artifacts but living systems that grow with their users. This shift from episodic training to continuous adaptation represents a fundamental change in how machine learning systems are conceived, moving closer to biological models of intelligence that learn throughout their operational lifetime.

Chapter 7

Conclusion

The whole is greater than the sum of its parts.

Aristotle

7.1 Summary of Contributions

This dissertation addresses fundamental architectural deficiencies in contemporary agent systems that prevent their deployment as trustworthy cognitive cores for Ambient Intelligence environments. Current frameworks suffer from three interconnected gaps: the Security Gap arising from ambient authority in execution environments, the Flexibility Gap preventing deployment across heterogeneous hardware platforms, and the Personalization Gap limiting adaptation to individual user contexts. These gaps are not isolated engineering challenges but symptoms of a fragmented paradigm that prioritizes compositional convenience over architectural coherence. The central thesis advanced and validated through this work is that trustworthy embodied agents demand vertical integration across the entire computational stack, from foundation model through execution substrate to adaptation mechanisms.

Addressing RQ1 The dissertation's primary contribution is a complete architectural blueprint demonstrating that secure, adaptable, and privately

personalized agents are achievable through principled system design. This blueprint encompasses three integrated pillars. The Foundation pillar establishes model sovereignty through transparent development of a tiered family of language models at 0.6B, 8B, and 14B parameter scales, trained via a rigorous three-stage protocol combining domain-focused pretraining, reasoning enhancement, and long-context extension. These models, optimized specifically for agentic computation, provide auditable cognitive cores that eliminate dependency on opaque commercial APIs while enabling formal verification and mechanistic interpretability impossible with black-box systems.

Addressing RQ2 The Framework pillar introduces a formally specified computational substrate built on WebAssembly that implements capability-based security through mathematical rigor. By replacing ambient authority with explicit capability passing, the framework ensures that tool code executes with precisely scoped permissions, mathematically guaranteeing the absence of privilege escalation vulnerabilities. The substrate provides concrete implementation patterns for compound AI systems, including recursive workflows, multi-model orchestration, and iterative refinement, supported by a repository of 20 production-ready tools distributed as auditable source code. Performance analysis demonstrates that the security benefits justify the modest overhead compared to native execution, validating the framework's practical viability.

Addressing RQ3 The Adaptation pillar presents Neuroplastic PEFT, a novel continuous personalization method that replaces gradient-based fine-tuning with localized Hebbian update rules. By leveraging the agent's live stream of neural activations as training signal, the method enables on-device learning without curated datasets or backpropagation infrastructure. Empirical evaluation demonstrates 12.5× sample efficiency compared to supervised approaches, achieving convergence after only 5-7 real-time interactions while exhibiting 99.6% knowledge retention across existing capabilities. This gradient-free approach dissolves the traditional dichotomy between offline training and online deployment, enabling agents to continuously evolve with users.

Addressing RQ4 The Smart Couch embodiment provides critical validation of the complete architecture in a physically deployed, safety-critical application. Integration of the 0.6B foundation model with the WebAssembly substrate on an NVIDIA Jetson Orin platform demonstrates sub-200ms time-to-first-token latency for interactive natural language processing, 96.3% intent fulfillment rate across standardized command corpora, and 78.3 mean System Usability Scale score from structured user studies. The embodiment successfully implements autonomous health monitoring, proactive posture correction, and personalized comfort optimization, validating that compact, transparent models can deliver sophisticated multimodal reasoning when properly integrated within secure execution environments.

Beyond individual technical contributions, the work demonstrates a paradigm shift from episodic, cloud-dependent agents to continuous, edge-native intelligent systems. Vertical integration enables emergent properties impossible in compositional black-box architectures. Formal verification of security properties becomes feasible when tool implementations are transparent WebAssembly modules rather than opaque API calls. Gradient-free adaptation becomes practical when model architectures are accessible for modification rather than locked behind service boundaries. Privacy-preserving personalization becomes realizable when all processing occurs on-device rather than requiring transmission of sensitive data to third parties.

The implications extend across technical, economic, and ethical dimensions. Technically, on-device processing and local adaptation preserve user sovereignty over sensitive personal data, aligning with privacy-by-design principles increasingly demanded by regulation and user expectations. Economically, edge deployment eliminates per-token API costs and reduces operational expenses compared to cloud-dependent architectures, enabling sustainable long-term operation. Ethically, transparency and user control address fundamental concerns about AI agency and accountability, providing technical foundations for alignment research by making model behavior auditable and modifiable.

This work contributes to the broader goal of democratizing AI by enabling sophisticated intelligence to operate independently of cloud infrastructure. The architectural principles established—vertical integration,

capability-based security, and gradient-free adaptation—apply beyond domestic applications to healthcare, education, and industrial automation where privacy and reliability are paramount. The transparent development of foundation models challenges the prevailing paradigm of AI as a service, demonstrating that competitive capabilities can be achieved through careful engineering of smaller, auditable systems. This transparency enables scientific scrutiny, regulatory compliance, and user trust impossible with opaque commercial models.

The formal security framework addresses growing concerns about AI safety by providing mathematical rather than empirical guarantees. As AI systems assume greater autonomy, the ability to formally bound their capabilities becomes essential for deployment in safety-critical contexts. The neuroplastic adaptation mechanism reconciles the seemingly incompatible requirements of continuous learning and privacy preservation. By enabling on-device personalization, the work demonstrates that AI systems can evolve with users without surveillance capitalism's data extraction imperatives.

7.2 Limitations and Future Directions

While this dissertation demonstrates the feasibility and advantages of vertically integrated agent architectures, several limitations constrain the immediate applicability and generalizability of the approach. These limitations also illuminate productive directions for extending the work's impact and addressing remaining challenges.

Current Limitations The most significant limitation concerns upfront investment requirements. Developing transparent foundation models from scratch demands substantial computational resources, specialized expertise in large-scale model training, and access to curated training corpora. The three-stage training protocol for the 14B model required weeks of computation on multi-GPU clusters, representing costs prohibitive for individual researchers or small organizations. This capital barrier contrasts sharply with the immediate accessibility of commercial API services, creating tension between long-term sovereignty benefits and short-term deployment pragmatism. While this investment amortizes across deployments

and provides architectural control impossible with APIs, the barrier to entry remains substantial.

Neuroplastic PEFT, despite demonstrated advantages in sample efficiency and catastrophic forgetting resistance, operates within fundamental constraints. The method learns associations by adjusting low-rank projections within the base model's existing semantic space, meaning it cannot introduce entirely novel concepts or capabilities absent from pretraining. An agent cannot learn about entities, procedures, or domains completely outside its foundational knowledge through Hebbian adaptation alone. This limitation necessitates complementary mechanisms such as Retrieval-Augmented Generation for incorporating truly novel information. Additionally, learning dynamics depend critically on the quality and breadth of the frozen base model, as Hebbian updates can only strengthen existing weak associations or create new connections between concepts already represented.

The WebAssembly substrate imposes performance overhead compared to native execution, with WebAssembly-to-machine-code compilation and sandboxed memory access introducing latency on the order of 10-30% relative to optimized native implementations. While evaluation demonstrates this overhead remains acceptable for interactive applications, latency-critical scenarios with microsecond-level timing constraints may find the security-performance tradeoff unacceptable. The capability-based security model also introduces conceptual complexity, as developers must explicitly reason about capability propagation and privilege boundaries rather than relying on ambient authority's implicit permissions.

The dissertation focuses on single-agent systems operating in structured domestic environments, assuming environments with predictable state spaces, well-defined sensor and actuator interfaces, and single-user or small-group interaction patterns. The Smart Couch platform, while demonstrating real-world feasibility, represents a controlled environment. Extension to multi-agent coordination, unstructured environments with high-dimensional observation spaces, continuous action spaces, and complex interactions remains an open challenge. The 0.6B model's narrow knowledge scope, while appropriate for domain-specific furniture control, would prove inadequate for general-purpose household assistants requiring broad world knowledge. Scaling to more complex embodiments demands propor-

tionally larger models, increasing computational requirements and exacerbating deployment constraints.

The work prioritizes edge deployment over cloud-based architectures, accepting constraints on model size and computational resources in exchange for privacy preservation and operational autonomy. This choice reflects the assessment that trust in personal AI systems requires local control over data and computation. The security model assumes honest-but-curious users who may inadvertently trigger vulnerabilities through prompt injection or misuse but are not actively malicious. Protection against sophisticated adversarial attacks, while important, exceeds the current scope. The target platform—consumer-grade embedded systems exemplified by the NVIDIA Jetson series—provides GPU acceleration within power and cost constraints suitable for domestic deployment. Scaling to more constrained microcontroller platforms or more powerful server infrastructure would require architectural adjustments.

Future Research Directions The architectural foundations established in this dissertation open numerous directions for extending capabilities, addressing limitations, and exploring novel applications of vertically integrated agent systems.

Neuroplastic PEFT can be enriched through alternative biologically inspired plasticity rules. Bienenstock-Cooper-Munro (BCM) theory introduces sliding thresholds for potentiation and depression based on postsynaptic activity history, enabling more nuanced adaptation to non-stationary input distributions. Spike-Timing-Dependent Plasticity (STDP), while challenging to map directly onto transformer architectures designed for parallel computation, could inspire novel attention mechanisms where temporal contiguity influences learning. Investigating three-factor learning rules incorporating reward signals could enable agents to selectively strengthen associations correlated with positive outcomes while weakening those associated with negative feedback, moving toward reinforcement learning paradigms without requiring explicit value function approximation.

Federated neuroplastic learning represents a compelling direction for privacy-preserving multi-agent collaboration. Current Neuroplastic PEFT operates independently on each device, with learned adaptations remaining local to individual users. A federated protocol could enable agents to

share learned associations across a user population while preserving privacy through differential privacy mechanisms or secure aggregation protocols. Agents could benefit from collective experience, learning from patterns observed across many users without exposing individual interaction histories. This approach would accelerate personalization for new users by initializing with population-level priors while maintaining individual sovereignty over sensitive data.

The WebAssembly tool repository and context-augmented retrieval infrastructure enable automated tool synthesis through compositional code generation. Future work could develop systems where agents programmatically create new tools by identifying relevant algorithmic patterns from existing implementations, adapting parameters and prompts for novel scenarios, and synthesizing hybrid tools combining components from multiple sources. This meta-programming capability would dramatically accelerate agent development, allowing non-expert users to specify desired functionality in natural language and receive production-ready WebAssembly tools generated through LLM-guided composition.

Extending adaptation mechanisms to multi-modal learning remains largely unexplored. Current Neuroplastic PEFT focuses on language, but the principles of unsupervised Hebbian learning apply equally to visual, auditory, and haptic modalities. An agent observing that a user consistently adjusts lighting when expressing discomfort could learn associations between environmental conditions, verbal cues, and corrective actions across sensory modalities. This cross-modal learning could enable more robust and context-aware personalization, as agents would ground linguistic concepts in perceptual experiences similar to human cognitive development.

Formal verification of dynamically loaded tools represents a critical research direction for high-assurance systems. While the WebAssembly substrate provides memory safety and capability-based security, verifying that tool implementations satisfy correctness properties, terminate within bounded time, and respect resource constraints remains manual and error-prone. Developing automated verification tools that prove mathematical properties about WebAssembly modules would enable deployment in safety-critical domains like medical devices or industrial automation, where failures carry severe consequences.

7.3 Closing Remarks

The path to trustworthy embodied intelligence requires reimagining the entire stack from silicon to semantics as a unified, principled system. This dissertation demonstrates that the fragmentation characterizing current agent architectures is not an inevitable consequence of system complexity but a deliberate choice prioritizing compositional convenience over correctness. By vertically integrating foundation models, execution substrates, and adaptation mechanisms, the work establishes that secure, adaptable, and privately personalized agents are not merely aspirational but practically achievable through rigorous system design.

The Smart Couch embodiment stands as tangible proof that compact, transparent models can deliver sophisticated reasoning when properly integrated within secure frameworks. The system achieves interactive latency, high accuracy, and positive user reception while maintaining strong privacy guarantees and resistance to catastrophic forgetting during continuous adaptation. These results validate the dissertation's central thesis and provide an existence proof that alternatives to the dominant paradigm of cloud-dependent, black-box agents are not only possible but superior along multiple dimensions of trustworthiness.

Looking forward, the architectural blueprint presented here contributes to the long-term realization of Ambient Intelligence by addressing fundamental prerequisites of transparency, security, and user control. As AI systems become increasingly integrated into intimate personal spaces, the ethical imperative for user sovereignty over data and algorithmic behavior intensifies. Vertically integrated architectures provide technical foundations for this sovereignty, enabling users to audit, modify, and control the intelligent systems serving them rather than being subject to opaque commercial services beyond their comprehension or influence.

This work reimagines the relationship between humans and AI from one of dependence on remote services to partnership with local intelligence. The Smart Couch, while humble in appearance, embodies a profound shift: AI as a personal capability rather than a corporate service, adapting to individual needs while respecting fundamental rights to privacy and autonomy. Rather than monolithic services owned by a handful of corporations, the vision is one of ecosystems comprising specialized

agents—transparently implemented, formally verified, and continuously adapting to serve individual users while preserving privacy and enabling collective benefit through federated learning.

The call to action for the research community is to prioritize architectural coherence and principled design over incremental feature additions to existing fragmented systems. Progress toward trustworthy AI demands not merely better models or more sophisticated prompting techniques but fundamental rethinking of how models, frameworks, and adaptation mechanisms compose into coherent wholes. The dissertation demonstrates that such rethinking yields systems with emergent properties—formal verification, gradient-free adaptation, and privacy preservation—impossible in black-box compositions.

Building agents worthy of human trust requires more than algorithmic sophistication. It demands architectural integrity, where security, adaptability, and privacy are not afterthoughts bolted onto existing systems but foundational principles informing every design decision from model architecture through execution substrate to adaptation protocols. The complete system presented in this dissertation proves that such integrity is achievable, providing a path forward for embodied intelligence that respects both human values and technical rigor. The journey from concept to couch validates not merely a specific implementation but a paradigm: vertical integration as the essential foundation for trustworthy autonomous agents in personal spaces.

Bibliography

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[2] Daniel Abowd, Anind K Dey, Robert Orr, and Jason Brotherton. Context-awareness in wearable and ubiquitous computing. *Virtual Reality*, 3(3):200–211, 1998.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[5] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[6] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. Guidelines for human-ai interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*, pages 1–13, 2019.

[7] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl, and Geoffrey E Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[8] William Ashby. *Design for a brain: The origin of adaptive behaviour*. Springer Science & Business Media, 2013.

[9] William Ross Ashby. An introduction to cybernetics. 1956.

[10] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[11] Amay J Bandodkar and Joseph Wang. Non-invasive wearable electrochemical sensors: a review. *Trends in biotechnology*, 32(7):363–371, 2014.

[12] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *HotSec*, 2006.

[13] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[15] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.

[16] Lucas Bourtoule, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *2021 IEEE symposium on security and privacy (SP)*, pages 141–159. IEEE, 2021.

[17] John Brooke. System usability scale (sus): a quick-and-dirty method of system evaluation user information. *Reading, UK: Digital equipment co ltd*, 43:1–7, 1986.

[18] Rodney A Brooks. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2):3–15, 1990.

[19] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[20] John Seely Brown, Allan Collins, and Paul Duguid. Situated cognition and the culture of learning. *1989*, 18(1):32–42, 1989.

[21] John Seely Brown and Paul Duguid. *The social life of information: Updated, with a new preface*. Harvard Business Review Press, 2017.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[23] Vannevar Bush et al. As we may think. *The atlantic monthly*, 176(1):101–108, 1945.

[24] Natalia Caporale and Yang Dan. Spike timing–dependent plasticity: a hebbian learning rule. *Annu. Rev. Neurosci.*, 31(1):25–46, 2008.

[25] Stuart K Card. *The psychology of human-computer interaction*. Crc Press, 2018.

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

[27] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[28] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PmLR, 2020.

[29] Yanjiao Chen, Baolin Zheng, Zihan Zhang, Qian Wang, Chao Shen, and Qian Zhang. Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. *ACM Computing Surveys (CSUR)*, 53(4):1–37, 2020.

[30] Andy Clark. *Being there: Putting brain, body, and world together again*. MIT press, 1998.

[31] Edmund M Clarke, Thomas A Henzinger, and Helmut Veith. Introduction to model checking. In *Handbook of Model Checking*, pages 1–26. Springer, 2018.

[32] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[33] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[34] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[35] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[36] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[37] Badhan Chandra Das, M Hadi Amini, and Yanzhao Wu. Security and privacy challenges of large language models: A survey. *ACM Computing Surveys*, 57(6):1–39, 2025.

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[39] Daniel C Dennett. *The intentional stance*. MIT press, 1989.

[40] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[41] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[42] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35:30318–30332, 2022.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[44] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction*, 16(2-4):97–166, 2001.

[45] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[46] Hebb Do. The organization of behavior. *New York*, 1949.

[47] Yanjie Dong, Haijun Zhang, Chengming Li, Song Guo, Victor Leung, and Xiping Hu. Fine-tuning and deploying large language models over edges: Issues and approaches. *arXiv preprint arXiv:2408.10691*, 2024.

[48] Paul Dourish. *Where the action is: the foundations of embodied interaction*. MIT press, 2001.

[49] Paul Dourish. The appropriation of interactive technologies: Some lessons from placeless documents. *Computer Supported Cooperative Work (CSCW)*, 12(4):465–490, 2003.

[50] Hubert L Dreyfus. *What computers still can't do: A critique of artificial reason*. MIT press, 1992.

[51] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International conference on machine learning*, pages 5547–5569. PMLR, 2022.

[52] Cynthia Dwork. Differential privacy. In *International colloquium on automata, languages, and programming*, pages 1–12. Springer, 2006.

[53] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and trends® in theoretical computer science*, 9(3–4):211–407, 2014.

[54] Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. Boosting and differential privacy. In *2010 IEEE 51st annual symposium on foundations of computer science*, pages 51–60. IEEE, 2010.

[55] Saad El Jaouhari and Eric Bouvet. Secure firmware over-the-air updates for iot: Survey, challenges, and discussions. *Internet of Things*, 18:100508, 2022.

[56] Gasser Elbanna, Neil Scheidwasser-Clow, Mikolaj Kegler, Pierre Beckmann, Karl El Hajal, and Milos Cernak. Byol-s: Learning self-supervised speech representations by bootstrapping. In *HEAR: Holistic Evaluation of Audio Representations*, pages 25–47. PMLR, 2022.

[57] Douglas C Engelbart. Augmenting human intellect: A conceptual framework. In *Augmented education in the global age*, pages 13–29. Routledge, 2023.

[58] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[59] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[60] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[61] Daniel Y Fu, Tri Dao, Khaled K Saab, Armin W Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

[62] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[63] Antonio Ginart, Melody Guan, Gregory Valiant, and James Y Zou. Making ai forget you: Data deletion in machine learning. *Advances in neural information processing systems*, 32, 2019.

[64] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78(110):1–108, 1998.

[65] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[66] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International journal of computer vision*, 129(6):1789–1819, 2021.

[67] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[68] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. More than you've asked for: A comprehensive analysis of novel prompt injection threats to application-integrated large language models. *arXiv preprint arXiv:2302.12173*, 27, 2023.

[69] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pages 79–90, 2023.

[70] Jonathan Grudin. From tool to partner: The evolution of human-computer interaction. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–3, 2018.

[71] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

[72] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35:35971–35983, 2022.

[73] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 185–200, 2017.

[74] Patrick J Hayes. The frame problem and related problems in artificial intelligence. In *Readings in artificial intelligence*, pages 223–230. Elsevier, 1981.

[75] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[76] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[77] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[78] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[79] Douglas R Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*. Basic books, 1999.

[80] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR, 2019.

[81] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[82] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pages 9118–9147. PMLR, 2022.

[83] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[84] Forrest N Iandola, Albert E Shaw, Ravi Krishna, and Kurt W Keutzer. Squeezebert: What can computer vision teach nlp about efficient neural networks? *arXiv preprint arXiv:2006.11316*, 2020.

[85] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research*, 24(251):1–43, 2023.

[86] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[87] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38, 2023.

[88] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[89] Qi Jing, Athanasios V Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. Security of the internet of things: perspectives and challenges. *Wireless networks*, 20(8):2481–2501, 2014.

[90] Leela S Karumbunathan. Nvidia jetson agx orin series. *A Giant Leap Forward for Robotics and Edge AI Applications. Technical Brief*, 2022.

[91] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, 60(1):87–116, 2022.

[92] Cory D Kidd, Robert Orr, Gregory D Abowd, Christopher G Atkeson, Irfan A Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *International workshop on cooperative buildings*, pages 191–198. Springer, 1999.

[93] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Reply to huszár: The elastic weight consolidation penalty is empirically valid. *Proceedings of the National Academy of Sciences*, 115(11):E2498–E2498, 2018.

[94] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[95] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International conference on machine learning*, pages 3519–3529. PMlR, 2019.

[96] Dmitry Krotov and John J Hopfield. Unsupervised learning by competing hidden units. *Proceedings of the National Academy of Sciences*, 116(16):7723–7731, 2019.

[97] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. *Advances in Neural Information Processing Systems*, 35:30233–30249, 2022.

[98] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joey Gonzalez, Hao Zhang, and Ion Stoica. vllm: Easy, fast, and cheap llm serving with pagedattention. *See https://vllm. ai/(accessed 9 August 2023)*, 2023.

[99] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[100] Hugh C Lauer and Roger M Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[101] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[102] Jerome Y Lettvin, Humberto R Maturana, Warren S McCulloch, and Walter H Pitts. What the frog's eye tells the frog's brain. *Proceedings of the IRE*, 47(11):1940–1951, 2007.

[103] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[104] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[105] Wenhao Li, Yubin Xia, and Haibo Chen. Research on arm trustzone. *GetMobile: Mobile Computing and Communications*, 22(3):17–22, 2019.

[106] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[107] Q Vera Liao and Jennifer Wortman Vaughan. Ai transparency in the age of llms: A human-centered research roadmap. *arXiv preprint arXiv:2306.01941*, 2023.

[108] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7(1):13276, 2016.

[109] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[110] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. In *Forty-first International Conference on Machine Learning*, 2024.

[111] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pages 22631–22648. PMLR, 2023.

[112] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[113] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[114] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006.

[115] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

[116] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. *arXiv preprint arXiv:1710.06963*, 2017.

[117] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[118] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

[119] Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR, 2018.

[120] Mark Miller. *Robust composition: Towards a uni ed approach to access control and concurrency control*. Johns Hopkins University, 2006.

[121] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research . . . , 2003.

[122] Marvin Minsky. *Society of mind*. Simon and Schuster, 1986.

[123] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 2007.

[124] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[125] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997.

[126] Allen Newell. The knowledge level: presidential address. *AI magazine*, 2(2):1–1, 1981.

[127] Allen Newell and Herbert Simon. The logic theory machine–a complex information processing system. *IRE Transactions on information theory*, 2(3):61–79, 1956.

[128] Thanh Tam Nguyen, Thanh Trung Huynh, Zhao Ren, Phi Le Nguyen, Alan Wee-Chung Liew, Hongzhi Yin, and Quoc Viet Hung Nguyen. A survey of machine unlearning. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–46, 2025.

[129] Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2(3):4, 2018.

[130] Zhijie Nie, Richong Zhang, Zhongyuan Wang, and Xudong Liu. Code-style in-context learning for knowledge-based question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18833–18841, 2024.

[131] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

[132] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[133] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

[134] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[135] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[136] Judea Pearl. *Causality*. Cambridge university press, 2009.

[137] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[138] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 5:606–624, 2023.

[139] Aman Priyanshu, Yash Maurya, and Zuofei Hong. Ai governance and accountability: An analysis of anthropic's claude. *arXiv preprint arXiv:2407.01557*, 2024.

[140] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40, 2024.

[141] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.

[142] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[143] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[144] Paolo Remagnino and Gian Luca Foresti. Ambient intelligence: A new multidisciplinary paradigm. *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and humans*, 35(1):1–6, 2004.

[145] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[146] Frank E Ritter, Farnaz Tehranchi, and Jacob D Oury. Act-r: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, 10(3):e1488, 2019.

[147] Daniel M Russell, Norbert A Streitz, and Terry Winograd. Building disappearing computers. *Communications of the ACM*, 48(3):42–48, 2005.

[148] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25(27):79–80, 1995.

[149] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[150] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[151] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72, 2010.

[152] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[153] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2002.

[154] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[155] Rainer Schnell. Privacy-preserving record linkage. *Methodological developments in data linkage*, pages 201–225, 2015.

[156] Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021.

[157] Sander Schulhoff, Jeremy Pinto, Anaum Khan, L-F Bouchard, Chenglei Si, Svetlina Anati, Valen Tagliabue, Anson Liu Kost, Christopher Carnahan, and Jordan Boyd-Graber. Ignore this title and hackaprompt: Exposing systemic vulnerabilities of llms through a global scale prompt hacking competition. Association for Computational Linguistics (ACL), 2023.

[158] John R Searle. Minds, brains, and programs. *Behavioral and brain sciences*, 3(3):417–424, 1980.

[159] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[160] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[161] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[162] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[163] Significant Gravitas. AutoGPT.

[164] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[165] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017.

[166] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2998–3009, 2023.

[167] Hari Sowrirajan, Jingbo Yang, Andrew Y Ng, and Pranav Rajpurkar. Moco pretraining improves representation and transferability of chest x-ray models. In *Medical Imaging with Deep Learning*, pages 728–744. PMLR, 2021.

[168] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10. IEEE, 2021.

[169] Norbert Streitz, Achilles Kameas, and Irene Mavrommati. *The disappearing computer: interaction design, system infrastructures and applications for smart environments*, volume 4500. Springer, 2007.

[170] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020.

[171] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[172] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[173] Lucille Alice Suchman. *Plans and situated actions: The problem of human-machine communication*. Cambridge university press, 1987.

[174] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[175] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[176] Qwen Team et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2(3), 2024.

[177] Edward Tenner. The design of everyday things by donald norman. *Technology and Culture*, 56(3):785–787, 2015.

[178] Hideyuki Tokuda, LT Yang, M Guo, GR Gao, and NK Jha. Smart furniture: A platform for creating context-aware ubiquitous applications everywhere. In *EUC*, page 1112. Springer, 2004.

[179] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[180] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing test: Philosophical and methodological issues in the quest for the thinking computer*, pages 23–65. Springer, 2007.

[181] Gina G Turrigiano and Sacha B Nelson. Homeostatic plasticity in the developing nervous system. *Nature reviews neuroscience*, 5(2):97–107, 2004.

[182] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[183] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.

[184] Praneeth Vepakomma and Otkrist Gupta. Tristan swedish, and ramesh raskar. *Split learning for health: Distributed deep learning without sharing raw patient data*, page 8, 2018.

[185] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

[186] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A practical guide, 1st ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[187] Sandra Wachter, Brent Mittelstadt, and Luciano Floridi. Why a right to explanation of automated decision-making does not exist in the general data protection regulation. *International data privacy law*, 7(2):76–99, 2017.

[188] Haiyang Wang, Yue Fan, Muhammad Ferjad Naeem, Yongqin Xian, Jan Eric Lenssen, Liwei Wang, Federico Tombari, and Bernt Schiele. Tokenformer: Rethinking transformer scaling with tokenized model parameters. *arXiv preprint arXiv:2410.23168*, 2024.

[189] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.

[190] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[191] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[192] Marc Weiser. The world is not a desktop. *interactions*, 1(1):7–8, 1994.

[193] Mark Weiser. The computer for the 21 st century. *Scientific american*, 265(3):94–105, 1991.

[194] Mark Weiser and John Seely Brown. Designing calm technology. *PowerGrid Journal*, 1(1):75–85, 1996.

[195] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[196] Norbert Wiener. *Cybernetics or Control and Communication in the Animal and the Machine*. MIT press, 2019.

[197] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, MIT, 1971.

[198] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

[199] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[200] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[201] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in neural information processing systems*, 32, 2019.

[202] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[203] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

[204] Shoshana Zuboff. The age of surveillance capitalism. In *Social theory rewired*, pages 203–213. Routledge, 2023.

Author's publications

1. N. Patwardhan, S. Marrone, and C. Sansone. Transformers in the Real World: A Survey on NLP Applications. Information, 14(4), 2023.
2. Q. Nasim, N. Patwardhan, T. Maiti, S. Marrone, and T. Singh. VeerNet: Using Deep Neural Networks for Curve Classification and Digitization of Raster Well-Log Images. Journal of Imaging, 9(7), 2023.
3. N. Patwardhan, L. Marassi, M. Gravina, A. Galli, M. Zuccarini, T. Maiti, T. Singh, S. Marrone, and C. Sansone. Responsible and Reliable AI at PICUS Lab. Convegno Nazionale CINI sull'Intelligenza Artificiale (Ital-IA 2023), Rome, Italy, May 29-30, 2023.
4. Q. Nasim, N. Patwardhan, J. Ali, T. Maiti, S. Marrone, T. Singh, and C. Sansone. Digitizer: A Synthetic Dataset for Well-Log Analysis. 22nd International Conference on Image Analysis and Processing (ICIAP-23), Udine, Italy, September 11-15, 2023.
5. L. Marassi, N. Patwardhan, and F. Gargiulo. Can Justice Be a Measurable Value for AI? Proposed Evaluation of the Relationship Between NLP Models and Principles of Justice. The First Workshop on User Perspectives in Human-Centred Artificial Intelligence (HCAI4U), 2023.
6. N. Patwardhan, S. Shetye, L. Marassi, M. Zuccarini, T. Maiti, and T. Singh. Designing Human-Centric Foundation Models. The First Workshop on User Perspectives in Human-Centred Artificial Intelligence (HCAI4U), 2023.
7. F. Amato, D. Benfenati, E. Cirillo, G.M. De Filippis, M. Fonisto, A. Galli, S. Marrone, L. Marassi, V. Moscato, N. Patwardhan, and A. Moccardi. Advancements and Challenges in Generative AI: Architectures, Applications, and Ethical Implications. Convegno Nazionale CINI sull'Intelligenza Artificiale (Ital-IA 2024), Naples, Italy, May 29-30, 2024.

8. A. Moccardi, E. Cirillo, C. Davino, M. Fonisto, F. Gargiulo, R.C. Ghosh, O. Gupta, R. Jaiswal, R. La Rovere, L. Marassi, N. Patwardhan, G.M. Orlando, D. Benfenati, G.M. De Filippis, A.E. Pascarella, D. Russo, C. Russo, C. Tommasino, S. Marrone, F. Amato, A.M. Rinaldi, V. Moscato, and C. Sansone. Generative AI: Developments, Applications, and Responsible Practices. Ital-IA 2025, Trieste, Italy, June 23-24, 2025.

9. A. Galli, F. Amato, D. Carlini, A. Del Prete, S. Dutto, N. Patwardhan, S. Marrone, V. Moscato, G. Piantadosi, C. Sansone, and M. Valle. Bridging AI and Industry: Research and Innovation from the CINI-AIIS Lab at Federico II University. Ital-IA 2025, Trieste, Italy, June 23-24, 2025.

10. L. Marassi, N. Patwardhan, S. Marrone, and C. Sansone. Misinformation and Disinformation in AI: the PICUS Lab Experience. Ital-IA 2025, Trieste, Italy, June 23-24, 2025.

11. L. Marassi, N. Patwardhan, S. Marrone, and C. Sansone. Trust in NLP Models: Ethical Considerations on the Reckless Use of AI. In Advanced Neural Artificial Intelligence: Theories and Applications, pp. 357-363. Springer Nature Singapore, 2025.