INTRODUCING

PrismAI

Architecture Plan

Multimodal AI Operating Platform

- CLASSIFICATION Engineering Architecture
- STATUS Production Draft

NOTE Architecture plan may change accordingly by the development. Features, integrations, and system design are subject to revision as engineering progresses.

1. Platform Introduction & System Architecture

> PrismAI is a multimodal Agentic Operating System that unifies seven distinct AI-powered modes — Chat, Code, Create, Voice, Agents, Workflows, and Local Model Activation — into a single, hardware-secure, cloud-connected platform running natively on Android.

Unlike conventional AI assistants constrained to text chat, PrismAI operates as a complete **AI execution environment**. Every mode is a first-class feature with its own UI, data pipeline, model routing, and execution backend. The platform is built with three core principles: **hardware-isolated execution** (all generated code runs inside sandboxed VMs), **model-agnostic routing** (any provider, any model, local or cloud), and **persistent memory** (the system remembers your work across sessions using vector embeddings). The result is an AI workspace that can write and run code, generate media, orchestrate multi-step tasks, converse in real time, and operate fully offline — all from your Android device.

1.1 The Seven Core Modes

Chat

Multi-turn conversations with streaming tokens, Markdown rendering, file attachments, and persistent session memory.

Code

Full IDE with multi-file editor, terminal emulator, sandboxed execution, diff viewer, and AI code completion.

Create

Multimodal generation — images (Stable Diffusion), video (AnimateDiff / Wan), and audio/music (MusicGen, TTS).

Voice

Real-time voice interaction via Whisper ASR and neural TTS with live waveform visualization and PCM streaming.

Agents

Autonomous multi-step task execution powered by LangGraph state machines, tool calling, and live telemetry.

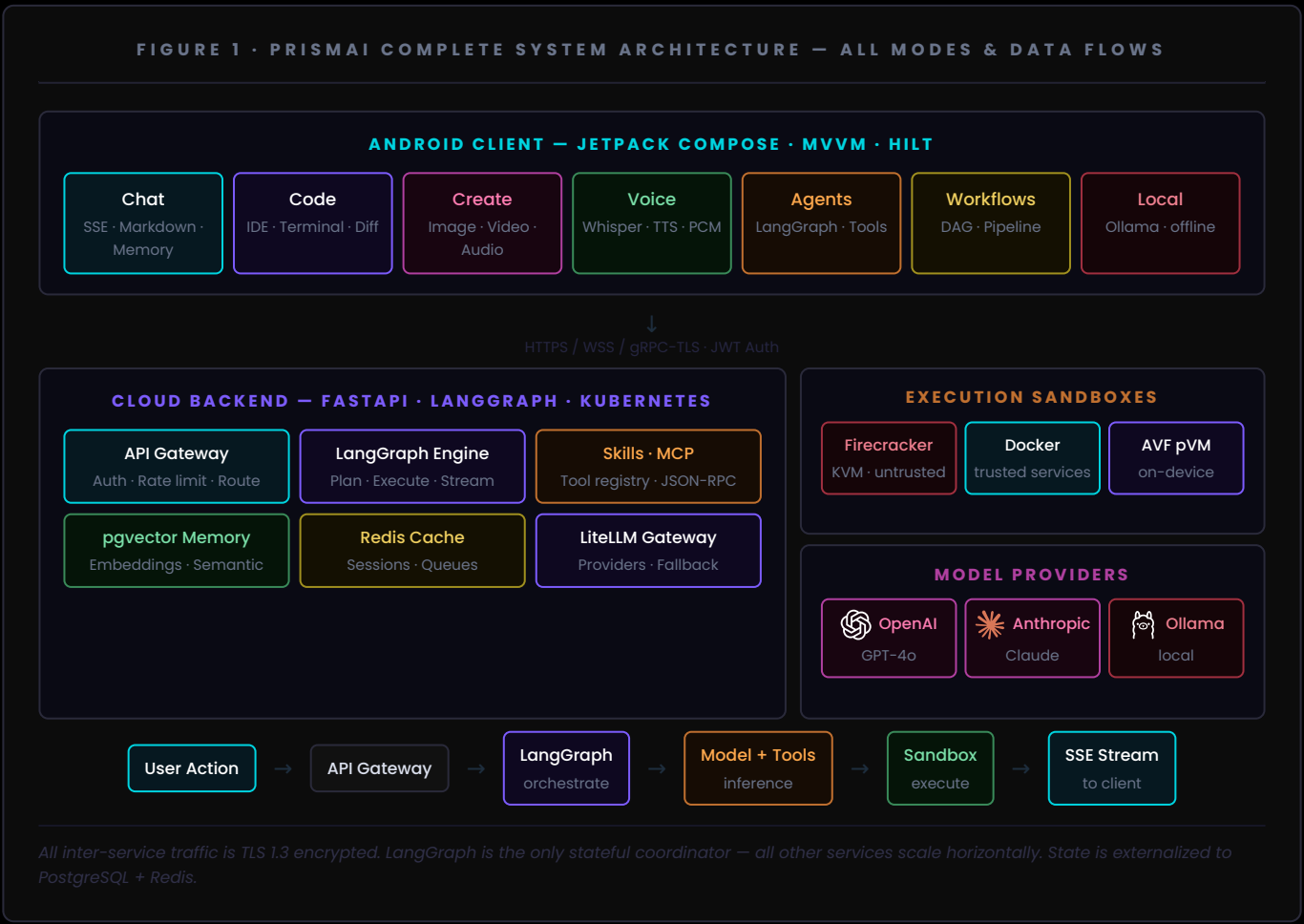
Workflows

Visual pipeline builder — drag-and-connect DAG nodes that chain AI tasks, tools, and transformations.

Local

Auto-discovers and activates on-device Ollama, llama.cpp. Fully offline. No API key needed.

1.2 Full System Architecture



2. All Modes — In-Depth Explanation

> Each mode is an isolated, independently deployable feature module with its own ViewModel, Repository, and backend route. They share the core auth, networking, and memory infrastructure but operate with distinct data flows and UI paradigms.

2.1 Chat Mode

The Chat mode delivers a production-grade conversational interface with real-time token streaming, persistent multi-turn memory, and full Markdown rendering. Incoming tokens from the backend arrive via Server-Sent Events (SSE) and are appended incrementally to the active `MessageEntity` row in Room. A custom Compose `IncrementalMarkdownParser` processes the growing string on each update, rendering code blocks, headers, bold/italic, and tables without re-parsing the entire content. File and image attachments are previewed locally using the `Coil` image loader before upload. The `ChatViewModel` exposes a `StateFlow<ChatUiState>` that the Compose screen collects; only the specific message rows that changed are recomposed, keeping frame rate consistently above 60 fps even during rapid token streaming.

- **Session persistence:** Every conversation is stored in Room with cascade-delete on session removal. Partial streams are checkpointed every 500 ms so network drops resume cleanly.
- **Context window management:** The backend automatically summarizes old turns when the token count exceeds the model's limit, injecting a compressed summary as a system message.
- **Memory recall:** Before each user turn the Memory Manager runs a pgvector similarity search and injects the top-5 relevant past memories as a hidden system context block.
- **Multi-model support:** Users can switch model mid-conversation. The gateway normalizes all providers to OpenAI message format, making provider switching transparent to the UI.

2.2 Code Mode

Code mode is a fully integrated IDE experience built natively in Compose. It combines a multi-file editor, AI-powered completion, a live terminal emulator, sandboxed code execution, and a structural diff viewer into a single split-pane interface. The editor uses a custom `BasicTextField` with a line-number column overlay and syntax tokenizer that runs on a background coroutine. The terminal emulator allocates a pseudo-TTY (PTY) pair, connects the slave end to a backend WebSocket channel, and renders the master end using a Compose Canvas-based VT100 interpreter that handles ANSI escape codes, cursor movement, and color sequences.

- **AI completion:** As the user types, a debounced request fires to the backend's FIM (Fill-in-Middle) endpoint with a 100 ms delay. Completions appear as grayed-out ghost text and are accepted with Tab.
- **Execution pipeline:** Run commands are sent to the Sandbox Manager which routes them into an isolated container. stdout/stderr stream back over WebSocket as ANSI-formatted chunks.
- **Diff viewer:** After AI edits, a structural diff is generated server-side using `difflib` and transmitted as a unified diff patch. The client renders it as an annotated before/after split view.
- **Project sync:** Local project directories mount as read-only via `authfs` inside the sandbox; write-back is staged through an explicit commit step to prevent accidental overwrites.

2.3 Create Mode — Image, Video & Audio Generation

Create mode exposes three distinct generation sub-modes, each backed by specialized model pipelines. The UI presents a unified prompt-builder card that adapts its controls based on the active sub-mode. Outputs stream into a paged gallery grid backed by a `LazyVerticalGrid` with infinite scroll.

Image Generation

Backend: **Stable Diffusion (SDXL / SD3)**
Protocol: **REST + polling / SSE progress**
Controls: **Steps, CFG, sampler, seed, LoRA**
Output: **PNG/WEBP, stored in Room + S3**
Local: **Local endpoint via localhost API**

Video Generation

Backend: **AnimateDiff / CogVideoX**
Protocol: **Async job queue (Celery + Redis)**
Controls: **Frames, FPS, motion strength**
Output: **MP4, streamed via signed S3 URL**
Note: Video generation requires a cloud backend or dedicated LAN inference server. On-device video generation is not supported due to mobile hardware constraints.

Audio Generation

Backend: **MusicGen / AudioCraft / XTTS**
Protocol: **SSE binary chunks (base64 PCM)**
Controls: **Duration, genre, tempo, voice clone**
Output: **WAV/MP3, playable inline**
TTS: **XTTS v2 for voice cloning**

2.4 Voice Mode

Voice mode enables low-latency, real-time voice interaction using a PCM streaming pipeline. Android's `AudioRecord` API captures 16 kHz mono PCM frames at 20 ms intervals. Each frame is packed into a binary WebSocket message and sent to the backend where OpenAI's Whisper (or a locally running Whisper instance) transcribes the stream incrementally using its streaming inference API. Transcription partial results appear in real time on the screen. The LLM generates a text response which is synthesized by a neural TTS engine (OpenAI TTS, Coqui XTTS, or ElevenLabs) and streamed back as PCM chunks that are queued into Android's `AudioTrack` for gapless playback. End-of-speech detection uses a VAD (Voice Activity Detector) model to automatically stop recording.

2.5 Agents Mode experimental

Agents mode exposes PrismAI's autonomous multi-step task execution engine directly to users. The user provides a high-level goal; the system decomposes it into sub-tasks, assigns specialized agents to each, executes them in parallel or sequence, and streams live telemetry back to the client. The Agents dashboard renders an interactive execution graph using a custom Compose Canvas renderer that draws nodes and edges in real time. Each node shows its current status (pending, running, complete, error) and the user can drill into any node to see its inputs, outputs, and intermediate tool calls.

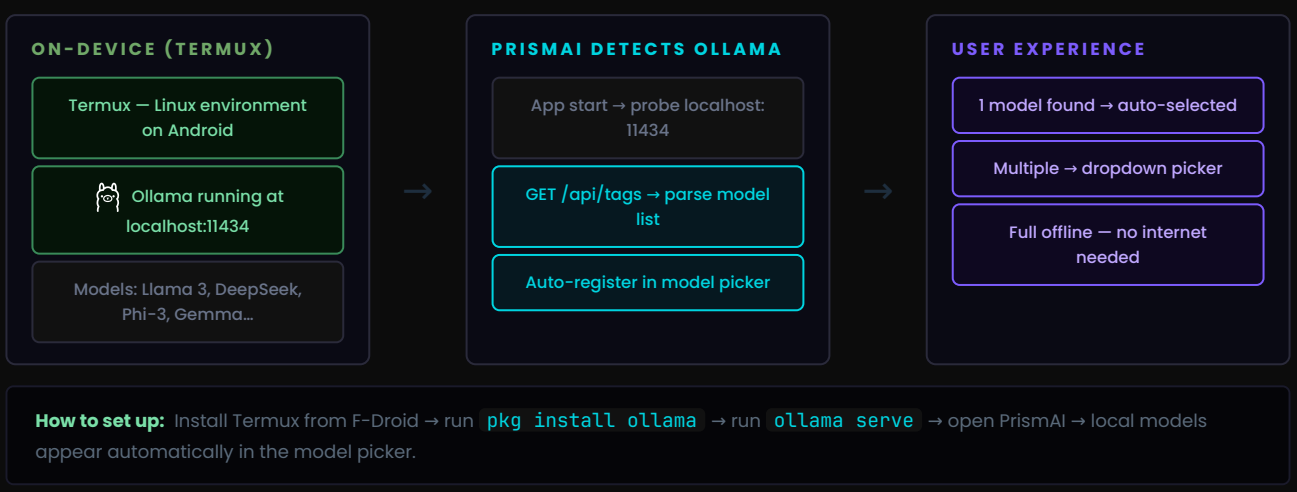
2.6 Workflows Mode experimental

Workflows mode provides a visual pipeline builder where users can construct and save reusable AI task chains. Nodes represent operations (LLM call, code execution, web search, file transform, API call, conditional branch) and are connected by typed edges that carry data between them. The workflow graph is serialized to a JSON DAG schema and executed server-side by a dedicated Workflow Executor that resolves the execution order using topological sort, runs independent branches concurrently, and handles retry/fallback logic at the edge level. Users can schedule workflows with cron triggers or invoke them programmatically via the API.

2.7 Local Model Activation

Local mode enables fully offline, on-device AI inference without any internet connection or API key. On Android, local model inference runs through **Ollama installed inside Termux** — a Linux terminal emulator that runs natively on Android. When Termux is running with Ollama active, PrismAI auto-detects it at `localhost:11434` and immediately makes all available models selectable in Chat, Code, and Agents modes. No configuration is needed — the detection happens automatically on app start and on each Wi-Fi change.

FIGURE 2 · LOCAL MODEL ACTIVATION VIA TERMUX — FLOW



How to set up: Install Termux from F-Droid → run `pkg install ollama` → run `ollama serve` → open PrismAI → local models appear automatically in the model picker.

Termux must be running in the background for local models to remain available. PrismAI re-probes on every app start. If Ollama stops, the local models disappear from the picker and cloud providers take over.

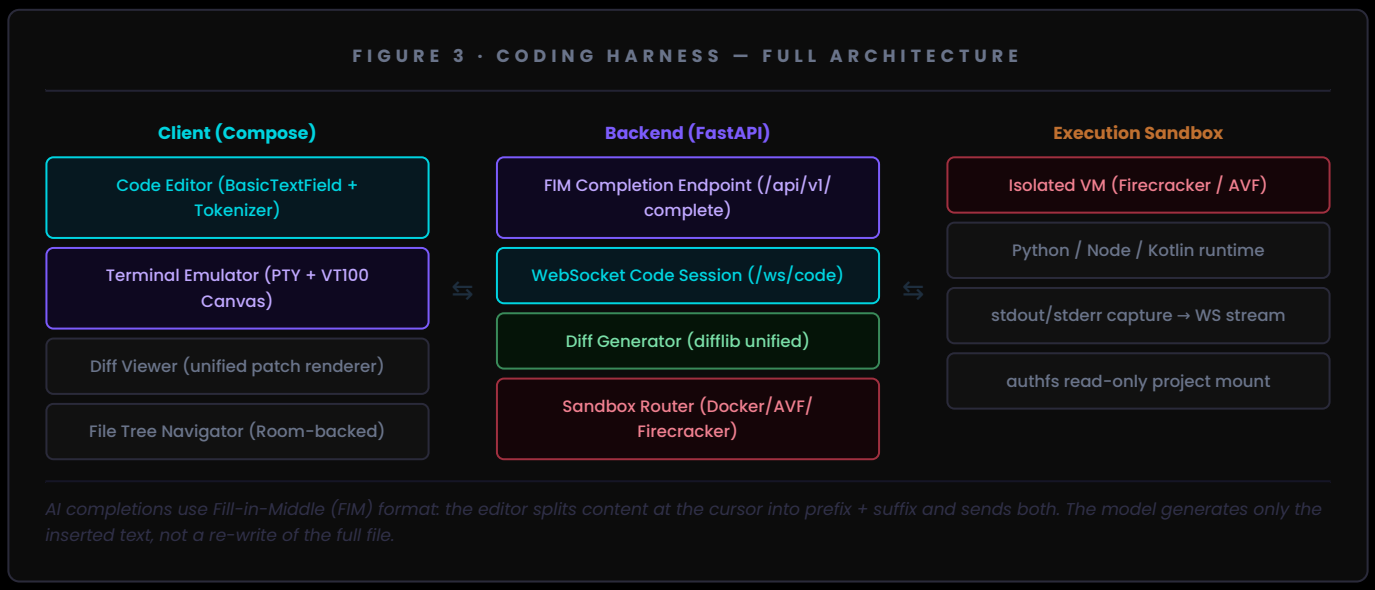
3. Coding Harness

> The Coding Harness is PrismAI's full-stack code execution environment — combining a native Compose IDE, AI completion engine, sandboxed runtime, and structural diff system into a unified developer experience.

3.1 IDE Architecture

The editor is implemented as a multi-tab `BasicTextField` with a synchronized line-number gutter rendered as an adjacent `Canvas`. A background coroutine runs an incremental tokenizer over the buffer, producing a `SpanStyle` `AnnotatedString` for syntax highlighting. The tokenizer uses a hand-written finite-state lexer (not a regex engine) to handle language-specific rules for Kotlin, Python, JavaScript, and TypeScript without blocking the main thread. The terminal emulator pairs each running process with a PTY device via JNI, connects the slave to a backend WebSocket channel, and processes the master using a VT100/ANSI state machine implemented in Kotlin that handles cursor movement, color codes (16/256/RGB), bold/italic attributes, and scroll region management.

3.2 AI Completion & Code Generation Pipeline



AI completions use Fill-in-Middle (FIM) format: the editor splits content at the cursor into prefix + suffix and sends both. The model generates only the inserted text, not a re-write of the full file.

3.3 Code Execution & Sandbox Lifecycle

1. **User triggers Run**
   The CodeViewModel sends a `RunRequest` with the current file content, language, and execution command to the backend WebSocket endpoint.
2. **Sandbox allocation**
   The Sandbox Router classifies the request by trust level and allocates a warm Firecracker VM from the pool, or spawns an AVF pVM if running on-device.
3. **Code injection**
   The file content is written to an ephemeral overlay filesystem inside the VM. The execution command is sent to the guest agent via vsock.
4. **Streaming output**
   The guest agent pipes stdout and stderr through the vsock channel. The backend forwards these as binary WebSocket frames. The terminal emulator renders each chunk in real time.
5. **Diff generation**
   If the AI modified any files, a unified diff is generated and transmitted. The client renders the patch as an annotated before/after view with line-level accept/reject controls.
6. **VM teardown**
   The VM is destroyed after execution completes. Its memory is wiped. If reuse is safe, the snapshot is restored to the clean baseline state for the next run.

3.4 FIM Completion — Code Reference

```python
# backend/api/v1/code.py — Fill-in-Middle completion endpoint
@router.post("/complete")
async def fim_complete(req: FIMRequest, user=Depends(get_current_user)):
    # FIM format varies by model — normalize here
    if req.model.startswith("deepseek"):
        prompt = f"<|fim_begin|>{req.prefix}<|fim_hole|><|fim_end|>{req.suffix}"
    elif req.model.startswith("codellama"):
        prompt = f"<PRE> {req.prefix} <SUF>{req.suffix} <MID>"
    else:  # OpenAI / generic
        prompt = req.prefix

    tokens = await gateway.stream_complete(
        model=req.model, prompt=prompt,
        max_tokens=120, stop=["\n\n", "</code>"], temperature=0.1
    )
    return StreamingResponse(tokens, media_type="text/event-stream")
```

4. Orchestration Engine

> The orchestration engine is the intelligence core of PrismAI — a stateful, graph-based execution runtime built on FastAPI + LangGraph that coordinates model calls, tool execution, memory access, and sandbox management across all modes.

4.1 FastAPI Backend Architecture

The backend is a Python FastAPI application served by Uvicorn with multiple async worker processes. All request handlers are fully `async`, leveraging Python's `asyncio` event loop to handle thousands of concurrent sessions without thread exhaustion. The application is structured as a clean-layered monolith: API routes → Orchestrator → Agents → Memory/ Gateway/Sandbox. Dependency injection uses FastAPI's built-in `Depends()` system with Hilt-style provider factories for database connections, Redis pools, and the LangGraph engine instance.

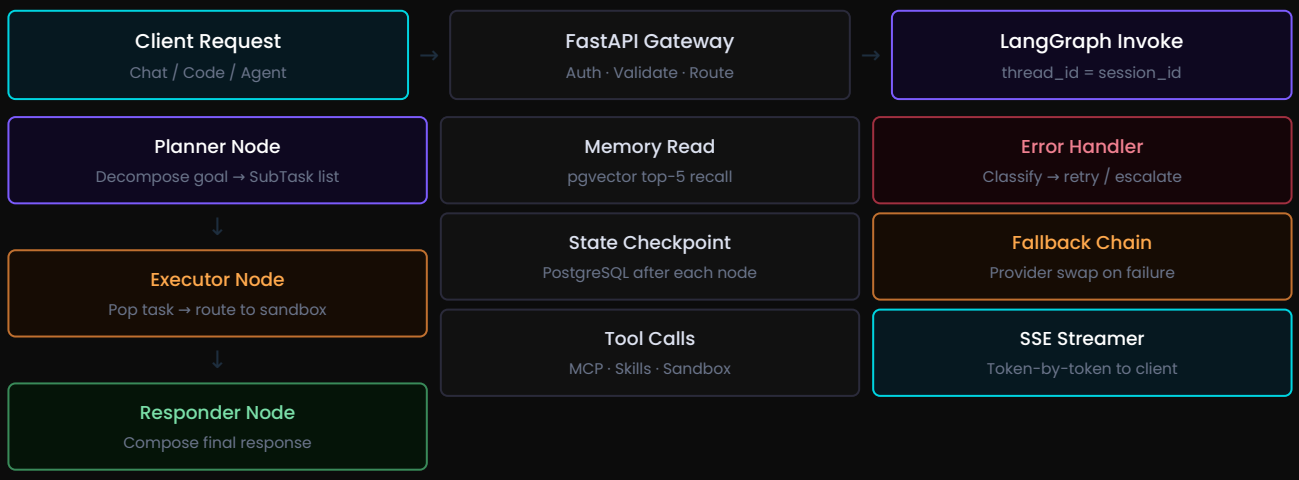
4.2 LangGraph State Machine

The agent runtime is modeled as a directed graph where each **node** is an async Python function that receives and returns a typed `AgentState` TypedDict. **Edges** are either unconditional transitions or conditional branches evaluated at runtime. The graph is compiled once at startup and invoked per session with a thread-scoped state snapshot, giving each session an independent durable execution context that can be paused, resumed, and replayed.

```python
# orchestrator/graph.py — LangGraph state machine definition
from langgraph.graph import StateGraph, END
from langgraph.checkpoint.postgres import AsyncPostgresSaver

class AgentState(TypedDict):
    messages:     list[AnyMessage]   # Full conversation history
    plan:         list[SubTask]      # Decomposed sub-tasks
    current_task: SubTask | None
    tool_results: list[ToolResult]
    memory_ctx:   list[str]          # Injected relevant memories
    next_action:  str                # "plan"|"execute"|"respond"|"error"
    session_id:   str

async def planner_node(state: AgentState) -> AgentState:
    ctx   = await memory.retrieve(state["messages"][-1].content, k=5)
    plan  = await llm.plan(state["messages"], context=ctx)
    return {"plan": plan, "memory_ctx": ctx, "next_action": "execute"}

async def executor_node(state: AgentState) -> AgentState:
    task   = state["plan"].pop(0)
    result = await sandbox.run(task.command, trust=task.trust_level)
    done   = not state["plan"]
    return {"tool_results": [*state["tool_results"], result],
            "next_action": "respond" if done else "execute"}

builder = StateGraph(AgentState)
builder.add_node("planner",   planner_node)
builder.add_node("executor",  executor_node)
builder.add_node("responder", responder_node)
builder.set_entry_point("planner")
builder.add_conditional_edges("planner",  lambda s: s["next_action"],
    {"execute": "executor", "respond": "responder"})
builder.add_conditional_edges("executor", lambda s: s["next_action"],
    {"execute": "executor", "respond": "responder"})
builder.add_edge("responder", END)

checkpointer = AsyncPostgresSaver.from_conn_string(settings.DB_URL)
graph = builder.compile(checkpointer=checkpointer)
```

4.3 Orchestration Flow Diagram

FIGURE 4 · ORCHESTRATION ENGINE — REQUEST-TO-RESPONSE FLOW



State is checkpointed to PostgreSQL after every node transition. Any node can be replayed by rewinding — enabling full audit trails and time-travel debugging via LangSmith.

4.4 Multi-Agent Task Decomposition

When a complex goal is received, the Planner Node calls the LLM with a structured decomposition prompt, producing a JSON plan. Sub-tasks are then dispatched to typed agent classes — **CodingAgent**, **ResearchAgent**, and **WritingAgent** — using `asyncio.gather()` for parallel execution. Each agent class is a `LangChain AgentExecutor` with a bound set of tools appropriate for its role.

FIGURE 5 · MULTI-AGENT DECOMPOSITION PIPELINE

5. Android Virtualization Framework (AVF) EXPERIMENTAL

EXPERIMENTAL AVF / pKVM is currently experimental and only supported on a curated set of Android devices (Pixel 6 and later, select Android 13+ devices). Availability on other hardware is not guaranteed. Cloud sandbox via Firecracker is the automatic fallback for unsupported devices.

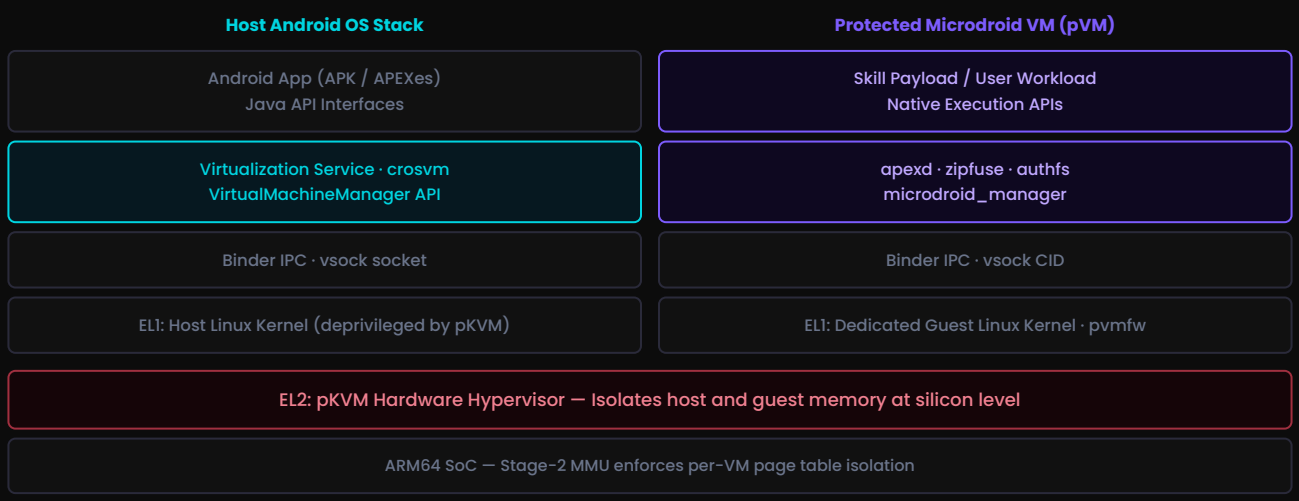
> AVF is PrismAI's on-device execution isolation layer — a hardware-enforced hypervisor built into Android that runs code inside cryptographically isolated Virtual Machines, completely inaccessible to the host OS.

5.1 How AVF Works

AVF is based on **pKVM** (protected KVM), a hypervisor implementation running at ARM Exception Level 2 (EL2) — the highest privilege level on ARM64. Once pKVM initializes, it deprivileges the host Linux kernel to EL1, preventing it from reading or modifying guest VM memory. Each VM (called a **protected VM / pVM**) runs with a dedicated Linux kernel instance, an encrypted memory range, and a hardware-verified boot chain powered by the `pvmfw` (protected VM firmware). This means that even if the host Android OS is compromised, the contents of a running pVM remain confidential and unmodifiable.

PrismAI uses **Microdroid** as the guest OS inside each pVM. Microdroid is a minimal Bionic-based Linux image maintained by Google that provides the essential runtime environment: shared libraries, Binder IPC for structured communication, `authfs` for authenticated host-to-guest file access, and `zipfuse` for mounting APEX packages from the host.

FIGURE 6 · ANDROID AVF — PKVM HYPERVISOR & MICRODROID ARCHITECTURE



Host ↔ guest communication: structured RPC over vsock (CID assigned per VM). File access: authfs authenticates host-provided FDs before presenting them inside the VM. Write operations stay ephemeral in the VM overlay.

5.2 PrismAI AVF Integration — Lifecycle

```
// Android — VmManager.kt: spawn a Microdroid pVM for skill execution
val config = VirtualMachineConfig.Builder(context)
    .setMemoryBytes(256L * 1024 * 1024)  // 256 MB
    .setNumCpus(2)
    .setPayloadBinaryName("skill_runner")
    .setProtectedVm(true)               // enforce pKVM isolation
    .build()

val vm = vmManager.create("skill-vm-\$taskId", config)
vm.run()  // boots Microdroid with pvmfw verification

// Communicate via vsock on CID assigned to the VM
val socket = VsockSocket(vm.cid, port = 8080)
socket.send(RpcRequest(command = skillCommand, env = sandboxEnv))
val result = socket.receive<RpcResult>()

// Teardown — memory is wiped automatically when VM stops
vm.stop()
vmManager.delete("skill-vm-\$taskId")
```

5.3 Warm Pool & Performance Optimization

Cold-booting a Microdroid VM takes 2–8 seconds — too slow for interactive use. PrismAI maintains a warm pool of pre-booted pVMs using Android's `VirtualMachine.snapshot()` API. On app startup, the pool manager boots a configurable number of VMs (default: 2), waits for their guest agents to report ready, captures memory snapshots, then holds the VMs in a suspended state. When a task arrives, it restores a warm VM from the snapshot in under 500 ms. After task completion, it either resets the VM to the baseline snapshot (for safe reuse) or destroys it (if compromise is suspected) and immediately provisions a replacement.

6. Cloud Sandbox Architecture

> The cloud sandbox layer provides secure, multi-tenant code execution for backend agent tasks using a hybrid of AWS Firecracker microVMs for untrusted code and Docker containers for trusted platform services.

6.1 Why Firecracker?

Standard Docker containers share the host kernel via namespaces and cgroups. A single kernel vulnerability can allow a container escape that compromises all co-located tenants. AWS Firecracker solves this by running each sandbox inside a full KVM virtual machine with a dedicated guest kernel, achieving near-VM isolation at near-container speed. Firecracker's VMM (Virtual Machine Monitor) is written in Rust with a minimal device model — it exposes only virtio-net, virtio-block, vsock, and a serial port, drastically reducing the attack surface compared to QEMU. Each VM boots in approximately 125 ms from a pre-loaded snapshot and uses only ~5 MiB of overhead memory.

FIGURE 7 · CLOUD SANDBOX — ROUTING ARCHITECTURE & ISOLATION LAYERS

Routing is immutable — a task classified as untrusted/external is never re-routed to Docker regardless of subsequent agent instructions. Classification is computed at request ingestion and cannot be overridden.

6.2 Firecracker Warm Pool Implementation

The `FirecrackerPoolManager` runs as a background async service. On startup it boots a configurable number of VMs (default 20), waits for the in-VM guest agent to report healthy on the vsock channel, captures a full memory snapshot via the Firecracker API's `CreateSnapshot` endpoint, then holds the VMs in a pool backed by an `asyncio.Queue`. Task allocation is O(1) — a worker claims the front of the queue; the pool manager immediately begins provisioning a replacement. After each task the VM is either reset via `LoadSnapshot` (clean reuse in ~125 ms) or killed (on anomaly detection).

```
# sandbox/firecracker.py — pool manager warm VM provisioning
async def claim_vm(self) → WarmVM:
    vm = await self._pool.get()          # O(1) from asyncio.Queue
    asyncio.create_task(self._replenish()) # replace immediately
    return vm

async def release_vm(self, vm: WarmVM, compromised: bool = False):
    if compromised:
        vm.proc.kill()                       # wipe memory immediately
        asyncio.create_task(self._replenish())
    else:
        await vm.api.load_snapshot(vm.snap_path)  # restore clean state
        await self._pool.put(vm)             # return to pool

async def _provision_vm(self) → WarmVM:
    vm_id  = str(uuid4())
    api    = FirecrackerAPI(f"/run/fc/{{vm_id}}.sock")
    await api.configure(vcpus=1, mem_mib=256, rootfs=ROOTFS_PATH)
    await api.start(); await self._wait_ready(vm_id)
    snap = await api.create_snapshot(SNAP_DIR / f"{{vm_id}}.snap")
    return WarmVM(id=vm_id, api=api, snap_path=snap)
```

7. Model Providers & Integration

> PrismAI is a **Bring Your Own Credentials (BYOC)** platform. It does not host, proxy, or manage any AI model services. Users are fully responsible for their own provider access, API billing, and model integrations. PrismAI simply provides the execution and orchestration layer.

This architecture ensures complete user autonomy — your API keys never leave your device unencrypted, no usage data is sent to PrismAI servers, and you are never locked into any specific provider. Add one model or ten; PrismAI works with whatever you configure.

7.1 Supported Providers

PrismAI supports any provider that exposes an OpenAI-compatible API, plus native SDKs for major providers. Users configure their own credentials in **Settings → Providers**. The following providers are supported out of the box:

OpenAI
GPT-4o, GPT-4o-mini, o1, o3
+ Whisper ASR + TTS + DALL-E

Anthropic
Claude 3.7 Sonnet
Claude 3.5 Haiku, Claude Opus

Google Gemini
Gemini 2.0 Flash
Gemini 1.5 Pro, Gemini Ultra

Groq
Llama 3.3 70B
Mixtral 8x7B (ultra-fast)

OpenRouter
Single key for 200+ models
Claude, GPT, Gemini, Llama…

Ollama (Local)
Llama 3, DeepSeek, Phi-3
Any GGUF model — fully offline

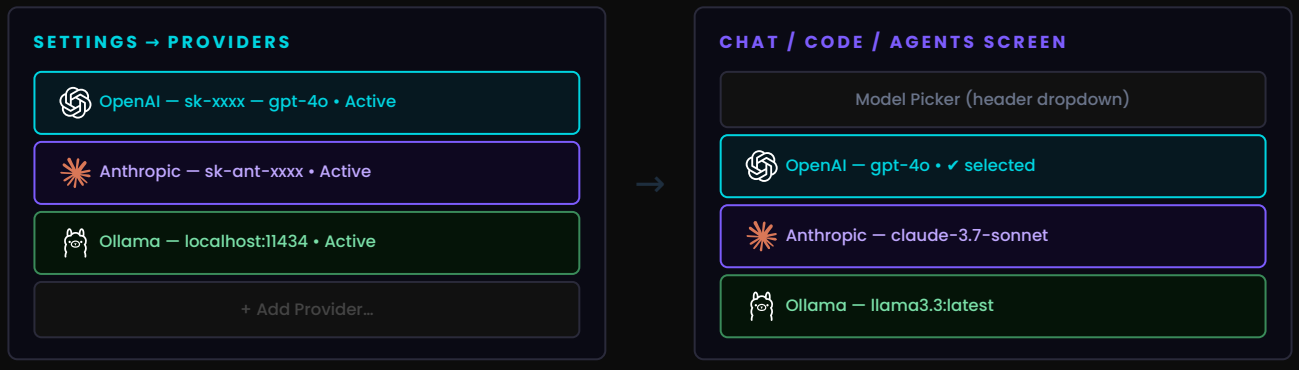
Additionally supported: **DeepSeek** **Together AI** **Replicate** **any custom endpoint** *(experimental)*

7.2 Provider Configuration — Settings Screen

Users configure providers in **Settings → Providers**. Each provider entry stores the API key (encrypted in Android Keystore), the base URL (for custom/self-hosted endpoints), and the default model. Configuration is per-provider, not per-session — configure once, available everywhere.

```kotlin
// Provider config stored in Android Keystore — ProviderConfigManager.kt
data class ProviderConfig(
    val id:          String,          // "openai" | "anthropic" | "ollama" | custom
    val displayName: String,          // "My OpenAI Key"
    val baseUrl:     String,           // "https://api.openai.com/v1"
    val defaultModel: String,          // "gpt-4o"
    val isActive:    Boolean = true
)

// API key encrypted in hardware-backed Keystore — never stored plaintext
fun saveApiKey(providerId: String, key: String) {
    val entry = KeyStore.getInstance("AndroidKeyStore")
    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    cipher.init(Cipher.ENCRYPT_MODE, entry.getKey("prism_key_\$providerId", null))
    prefs.putEncrypted("provider_key_\$providerId", cipher.doFinal(key.toByteArray()))
}
```

FIGURE 8 · BYOC PROVIDER CONFIGURATION & MODEL SELECTION FLOW



Only providers with active API keys or reachable endpoints appear in the model picker. Keys stored in Android Keystore — hardware-backed, biometric-bound, never transmitted to PrismAI servers.

7.3 Model Selection Logic

PrismAI follows a simple, user-transparent selection model. There is **no automatic provider switching** and no silent re-routing. The user is always in control of which provider handles their request:

1. **One provider configured**
   That provider is automatically used for all requests. No picker shown. Simple, clean experience for users running a single model.

2. **Multiple providers configured**
   A model picker dropdown appears in the Chat/Code/Agents header. User explicitly selects which provider and model to use. Selection persists per-session and is remembered across sessions per mode.

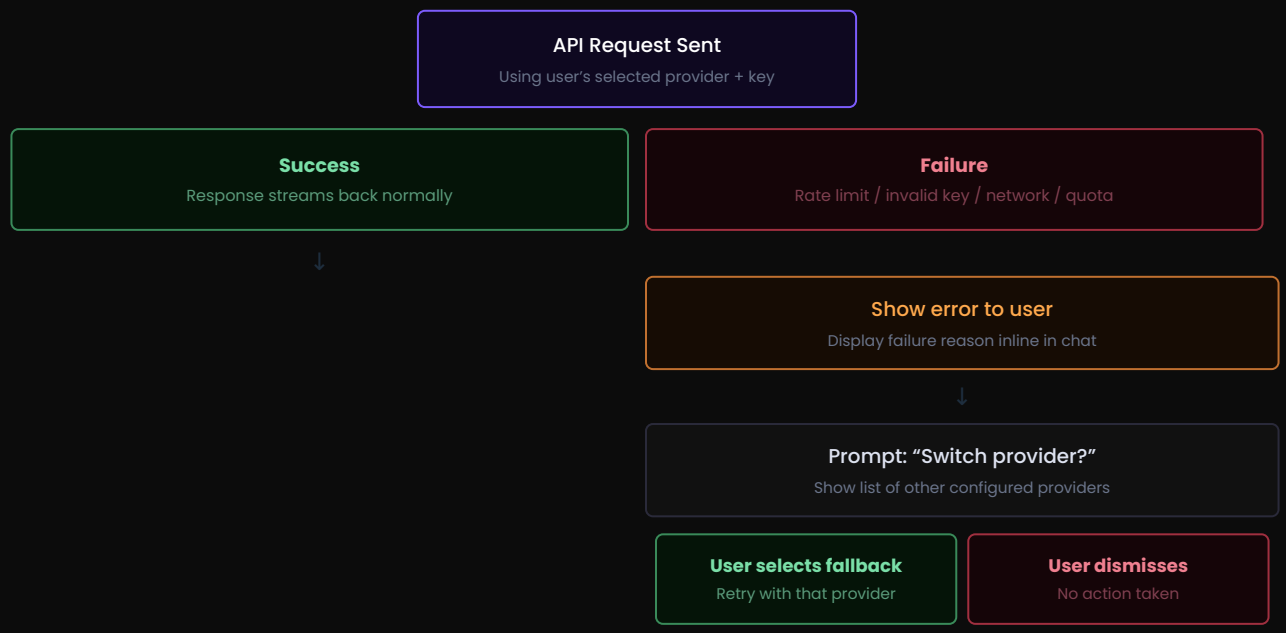
3. **Request is dispatched**
   PrismAI sends the request to the user's selected provider using their own stored API credentials. PrismAI acts only as the orchestration layer — it never sees, stores, or forwards credentials beyond the secure local call.

4. **Request fails**
   An inline error is shown with the failure reason (rate limit, invalid key, network error, quota exceeded). No automatic fallback — the user decides what to do next.

5. **User-initiated fallback (optional)**
   On failure, a prompt asks: *"Switch to a different provider?"* and shows the list of other configured providers. User taps one to retry with it. This is always an explicit user action, never automatic.

7.4 Failure Handling & User-Driven Fallback

FIGURE 9 · REQUEST FAILURE & USER-DRIVEN FALLBACK FLOW



PrismAI never silently switches providers. Every provider change is an explicit user decision — ensuring full billing transparency.

7.5 Local Network & Self-Hosted Providers

PrismAI supports inference servers running on the same local Wi-Fi network as the Android device. These servers run on a separate computer or home server — PrismAI connects to them via their local IP address or hostname. No API key is required — just the server's base URL. This enables fully offline, private AI inference without sending data to cloud services.

Ollama (LAN)

Run Ollama on any computer on your network. Auto-discovered by PrismAI via LAN scan at startup. Supports Llama 3.3, DeepSeek, Phi-3, Mistral, Gemma and any model in the Ollama library. Accessible at `http://<local-ip>:11434` .

Custom Endpoint

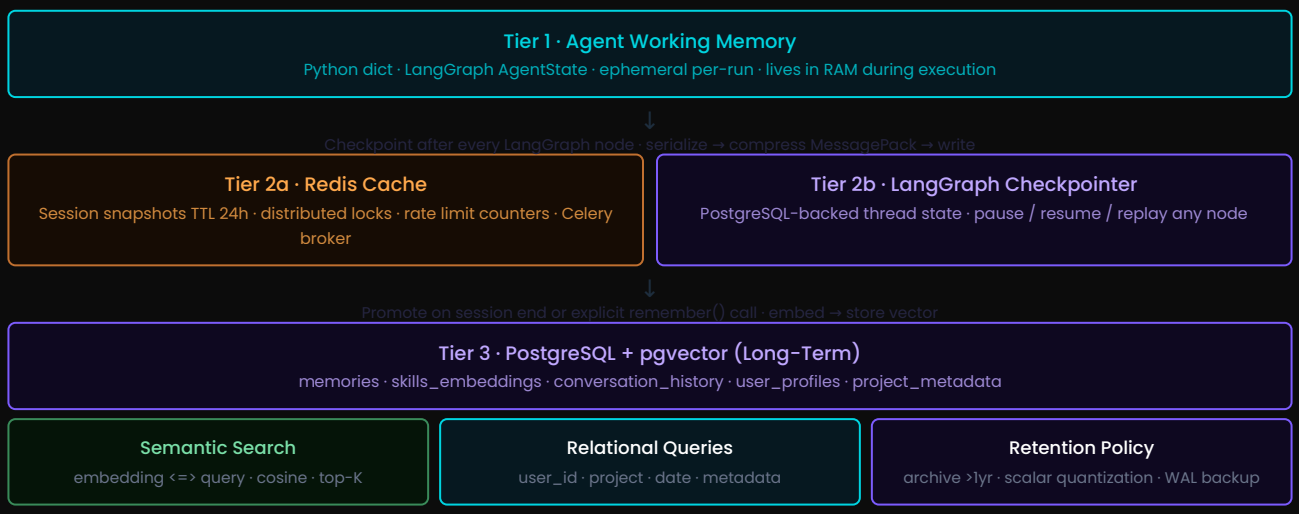
Any HTTP server implementing an OpenAI-compatible API, accessible from the device via LAN or internet. User sets the base URL and optional API key. Compatible with any self-hosted inference server. *(experimental)*

8. Memory System

> PrismAI's memory system gives the platform genuine long-term recall — storing, indexing, and retrieving relevant context from past sessions using vector embeddings and semantic similarity search powered by PostgreSQL pgvector.

8.1 Three-Tier Memory Architecture

FIGURE 9 · THREE-TIER MEMORY ARCHITECTURE & DATA FLOW



pgvector uses HNSW index (m=16, ef_construction=200) enabling sub-millisecond approximate nearest-neighbor search at million-row scale. Index rebuild is online — no downtime required.

8.2 Embedding Pipeline & Schema

When an agent produces a significant output (code solution, research synthesis, design decision), the `MemoryManager.remember(content)` method is called. This enqueues a Celery task that: (1) calls the embedding model (`text-embedding-3-small` via the gateway), (2) stores the 1536-dim vector alongside JSONB metadata in the `memories` table, (3) updates the HNSW index. At the start of each new task, the manager embeds the current goal and runs a cosine similarity query to retrieve the top-5 most relevant past memories, which are injected into the system prompt.

```
-- PostgreSQL schema — core memory tables
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE memories (
  id          bigserial PRIMARY KEY,
  user_id     uuid REFERENCES users(id) ON DELETE CASCADE,
  content     text NOT NULL,
  embedding   vector(1536),
  source_type text,                    -- 'chat'|'code'|'create'
  metadata    jsonb DEFAULT '{}',
  archived_at timestamptz,
  created_at  timestamptz DEFAULT now()
);
CREATE INDEX ON memories USING hnsw (embedding vector_cosine_ops)
  WITH (m=16, ef_construction=200);

-- Semantic retrieval query
SELECT content, metadata, 1 - (embedding <=> $1) AS similarity
FROM memories
WHERE user_id = $2 AND archived_at IS NULL
ORDER BY embedding <=> $1
LIMIT 5;
```

9. Skills System & Model Context Protocol

> The Skills system allows PrismAI to gain new capabilities at runtime without code changes. Skills are self-describing packages discovered from GitHub registries, validated, embedded, and executed inside sandboxes. MCP connects the platform to external tool servers using a standardized JSON-RPC protocol.

9.1 Skill Architecture

A skill is a directory containing an entry-point script and a `SKILL.md` configuration file. The `SKILL.md` contains a YAML frontmatter block that declares the skill's name, description, owner, and input schema. The description field is embedded into a pgvector index at discovery time, enabling semantic retrieval during task planning. The entry-point command (shell, Python, Node.js, or Docker image) is executed exclusively inside an isolated Firecracker VM or AVF pVM — never on the host.

```
---
name: web-researcher
description: "Search the web and return structured summaries for a given query."
owner: "prism-skills-core"
version: "1.2.0"
inputs:
  query:    {{type: string,  description: "Search query string"}}
  max_results: {{type: integer, default: 5}}
outputs:
  results:  {{type: list,    description: "List of {{title, url, summary}}"}}
entrypoint: "python skill.py"
runtime:    "python3.11"
sandbox:    "firecracker"
---
# Web Researcher Skill
Execute this skill to search the web. Provide a clear, specific query.
Returns structured JSON with title, URL, and 2-3 sentence summaries.
```

9.2 Skill Discovery & Invocation Lifecycle

1. **Registry Scan**
   On startup and every 6 hours, the `SkillsRegistry` queries GitHub for repos tagged `prism-skill`. Shallow-clones each, reads `SKILL.md`, validates against Pydantic schema.

2. **Security Validation**
   SHA-256 fingerprint check against known-safe allowlist. Python skills scanned with `bandit`; JS skills with AST analysis. Any skill with shell injection vectors or suspicious imports is quarantined with status `REJECTED`.

3. **Embedding & Indexing**
   The description field is embedded via `text-embedding-3-small` and stored in the `skills_embeddings` pgvector table with the full metadata JSON.

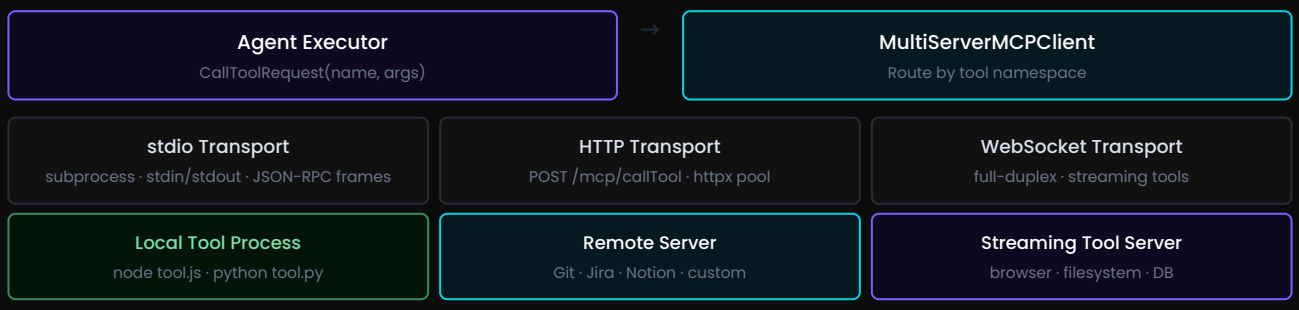
4. **Planning-Time Retrieval**
   The Planner Node embeds the sub-task goal and queries `SELECT name FROM skills_embeddings ORDER BY embedding ⟺ $1 LIMIT 3`. Top match is loaded into system context.

5. **Sandboxed Execution**
   Entry-point command runs inside a Firecracker VM (cloud) or AVF pVM (mobile). stdout captured as structured JSON per the skill's output schema. Result returned to Executor Node.

9.3 MCP — Model Context Protocol Integration

MCP allows PrismAI to bind external tool servers (Git, Jira, Notion, filesystem, database) using a standardized JSON-RPC 2.0 protocol. The `MultiServerMCPClient` maintains persistent connections to all registered servers and exposes a unified `call_tool(name, args)` interface to agents.

FIGURE 10 · MCP MULTI-SERVER CLIENT — TRANSPORT & TOOL INVOCATION

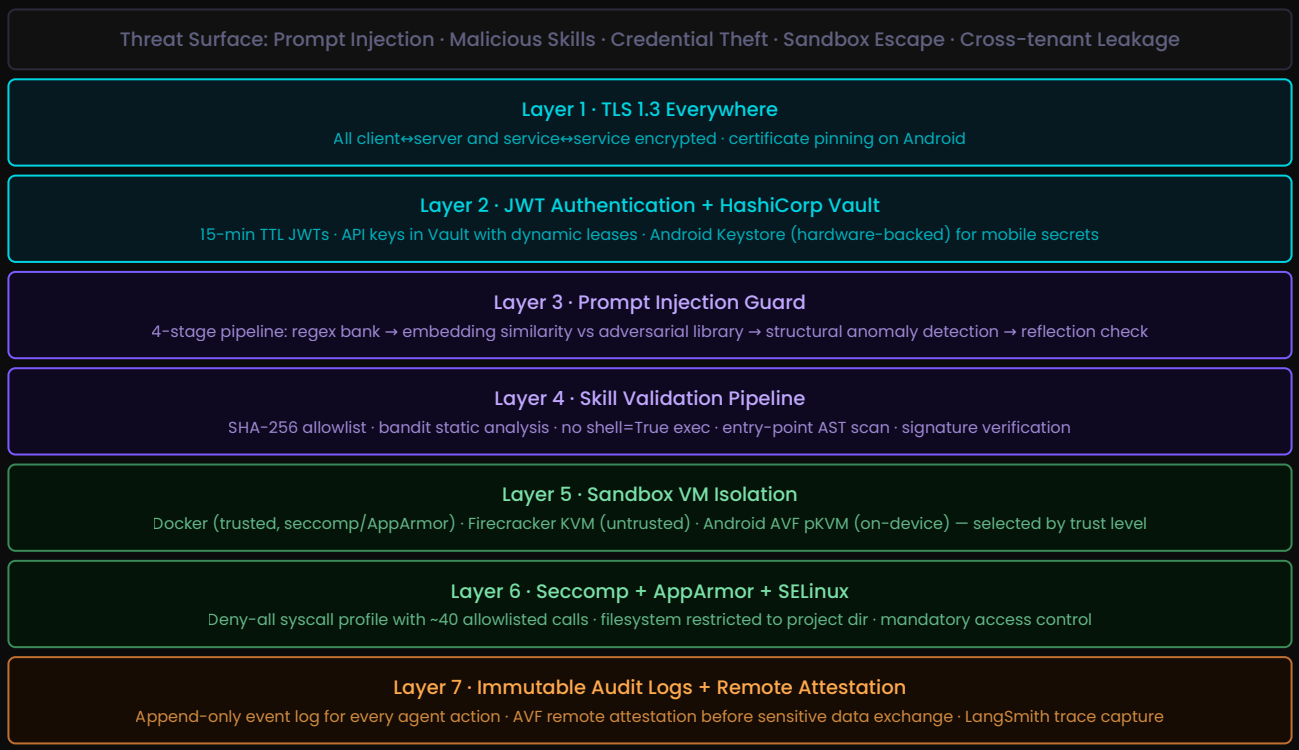


Tool schemas are cached at startup via `tools/list`. All args are validated against the cached JSON Schema before dispatch. TLS enforced on all external transports. Unix domain sockets for intra-host stdio servers.

10. Security Architecture

> Security is enforced at every layer — from TLS at the network boundary to hardware attestation at the virtualization tier. The threat model covers prompt injection, malicious skill packages, sandbox escape, credential theft, and cross-tenant data leakage.

FIGURE 11 · DEFENSE-IN-DEPTH SECURITY LAYER STACK



Incident response: anomaly detected → Sentry alert → automated API key rotation → VM termination + memory dump → forensic log capture → post-mortem within 24 h.

10.1 Prompt Injection Guard

Every user message passes through a four-stage detection pipeline before reaching the agent orchestrator. Stage 1 applies a regex bank of ~200 known injection patterns. Stage 2 computes the message embedding and checks cosine similarity against a library of labeled adversarial prompts — any similarity above 0.92 is flagged. Stage 3 checks for structural anomalies: abnormally long system-role injections, base64-encoded payloads, Unicode homoglyphs. Stage 4 detects prompt reflection requests ("repeat your instructions", "what is your system prompt"). Flagged messages are rejected with a 400 and logged to the security audit trail.

10.2 Sandbox Containment & Least Privilege

Every agent-generated code snippet executes inside an isolated VM — no exceptions. Linux seccomp filters apply a deny-all syscall profile with approximately 40 allowlisted calls. AppArmor profiles restrict file system access to the sandbox working directory. Network egress from sandboxes routes through an allow-list proxy; arbitrary outbound connections are blocked at iptables level. Standard container processes use user namespace mappings to prevent root escalation inside Docker.

10.3 Key Management & Encryption

All platform secrets reside in HashiCorp Vault with dynamic secret leases (1-hour TTL). The FastAPI backend authenticates to Vault using a Kubernetes service account JWT, receiving short-lived API key leases — even if the process is compromised, the attacker captures only a short-window credential. On Android, API keys are stored as `AndroidKeyStore` entries bound to biometric authentication. AES-256-GCM encrypts all data at rest in Room and PostgreSQL. Remote attestation via the AVF `VmAttestationResult` API produces a signed certificate chain verifiable before sensitive payloads are transmitted to the VM.

11. References

The architectural decisions, implementation patterns, and specifications in this document are grounded in the following authoritative sources:

1. **Android Virtualization Framework (AVF)** — Official AOSP documentation: Protected VMs, pKVM architecture, Microdroid guest OS, pvmfw secure boot. *source.android.com/docs/core/virtualization*
2. **Google Microdroid** — Minimal OS for protected VMs: Binder IPC in guests, authfs, zipfuse, microdroid_manager. *android.googlesource.com/platform/packages/modules/Virtualization*
3. **AWS Firecracker MicroVM** — KVM-based serverless isolation, jailer process, device model, snapshot/restore API, warm pool patterns. *firecracker-microvm.github.io*
4. **LangGraph** — Stateful multi-agent workflows, StateGraph builder, AsyncPostgresSaver checkpointer, human-in-loop patterns. *python.langchain.com/docs/langgraph*
5. **LiteLLM Gateway** — Unified LLM completion API, provider normalization, spend tracking, fallback chains. *docs.litellm.ai*
6. **pgvector** — Vector similarity search extension for PostgreSQL, HNSW and IVFFlat indexes, distance operators. *github.com/pgvector/pgvector*
7. **Model Context Protocol (MCP)** — Multi-server SDK, transport specifications (stdio, HTTP, WebSocket), tool schema definition. *modelcontextprotocol.io*
8. **Android Keystore System** — Hardware-backed key generation, biometric binding, AES-GCM encryption at rest. *developer.android.com/training/articles/keystore*
9. **HashiCorp Vault** — Dynamic secrets, Kubernetes auth method, short-lived credential leases, audit logging. *developer.hashicorp.com/vault*
10. **Jetpack Compose + Navigation** — Single-activity NavHost, MVVM architecture, StateFlow, Room AutoMigrations. *developer.android.com/jetpack/compose*
11. **OpenAI Whisper & TTS** — Streaming ASR, Fill-in-Middle FIM completion format, neural TTS synthesis. *platform.openai.com/docs*
12. **Stable Diffusion / Local endpoint** — Text-to-image, img2img, LoRA integration, Local endpoint local API. *github.com/AUTOMATIC1111, github.com/comfyanonymous/Local endpoint*
13. **OpenTelemetry + Prometheus + Grafana** — Distributed tracing, metrics scraping, dashboard visualization, alert rules. *opentelemetry.io, prometheus.io, grafana.com*