Agent Arvyn v2.1: A Neuro-Symbolic Large Action Model (LAM) for Autonomous Financial Orchestration

Abstract

The current trajectory of Artificial Intelligence has been defined by the rapid maturation of Large Language Models (LLMs) which excel in generative tasks but often falter in kinetic execution. This limitation, known as the "Execution Gap," is particularly pronounced in high-stakes domains such as autonomous financial orchestration, where the cost of error is non-negotiable. **Agent Arvyn v2.1** represents a definitive architectural response to this challenge. It is not merely a conversational interface but a **Type-2 Large Action Model (LAM)** designed to navigate, perceive, and manipulate the rigid, asynchronous interfaces of modern banking infrastructure.

This blueprint delineates the comprehensive technical architecture for Arvyn v2.1, explicitly modifying the v2.0 proposal to integrate **Google's Gemini 1.5 Flash** as a unified multimodal cognitive engine. This shift consolidates Speech Recognition (STT) and Intent Logic into a single, low-latency inference step, significantly reducing the "time-to-action" while enhancing semantic understanding. Simultaneously, the architecture strictly adheres to a **Zero Trust** output philosophy by utilizing **Windows Native Text-to-Speech (TTS)**, ensuring that sensitive financial confirmations are synthesized locally and never transmitted to the cloud. Furthermore, the system introduces a deterministic **"Click-to-Start/Click-to-Process"** interaction model, a deliberate UX pivot designed to verify user presence and intent physically, thereby eliminating the security vulnerabilities inherent in "always-listening" wake-word systems.

Specifically targeted at the architecture of the **Rio Finance Bank** portal, this document details the selector strategies, DOM hydration logic, and state management required to automate a Single Page Application (SPA). By synthesizing the probabilistic reasoning of Gemini with the deterministic execution of Microsoft Playwright, Arvyn v2.1 establishes a robust, privacy-first standard for personal financial agents.

---

1. The Neuro-Symbolic Paradigm Shift

1.1 Redefining Agency: From Conversationalists to Operators

The contemporary landscape of AI assistants is dominated by "Chatbots"—systems constrained to the generation of text. While adept at answering queries, these agents are

fundamentally disconnected from the operational layer of the digital world. A user can ask a chatbot how to pay a bill, but the chatbot cannot *pay* the bill. This forces the user to remain the "human router," translating the agent's advice into manual clicks and keystrokes.

Agent Arvyn v2.1 rejects this passive role. It is architected as a **"Do-Bot,"** a kinetic extension of the user's will. Its primary operational domain is the brittle environment of web banking, where interfaces are designed for human eyes, not API calls. To bridge this gap, Arvyn employs a **Neuro-Symbolic Architecture**, a hybrid methodology that combines the strengths of two distinct branches of AI:

1. **The Neuro Component (Intuition):** Powered by **Gemini 1.5 Flash**, this layer handles the messy, unstructured inputs of the real world—voice commands filled with hesitation, varying accents, and implicit intent. It provides the "probabilistic intuition" required to understand that a user saying "Clear my electricity dues" maps to the "Bill Pay" button, even if the button is labeled "Utility Payments."
2. **The Symbolic Component (Precision):** Powered by **Microsoft Playwright**, this layer executes the actual interaction. Unlike the neural layer, which deals in probabilities, the symbolic layer deals in absolutes. It executes deterministic code paths: locate element x, verify visibility, click, wait for navigation.

This duality ensures resilience. The Neuro brain allows the agent to adapt to changes in the banking UI (e.g., a button moving location), while the Symbolic hands ensure that the transaction is executed with mathematical precision, impossible for a purely generative model to guarantee.

1.2 The Privacy-Performance Optimization Curve

A critical design constraint for Arvyn v2.1 is the balance between the computational power required for understanding and the privacy required for financial data.

In the v2.0 proposal, a local SLM (Small Language Model) was suggested. However, benchmarks indicate that local models often struggle with the complex disambiguation required for banking contexts (e.g., distinguishing "Current Account" from "Savings Account" based on vague user phrasing) and introduce significant latency on consumer hardware.

Arvyn v2.1 solves this by offloading the *input processing* to the cloud while keeping the *output synthesis* local.

- **Input (Gemini 1.5 Flash):** The user's voice command is encrypted and sent to Google's enterprise-grade Gemini API. This allows us to leverage a massive context window (1 million tokens) and state-of-the-art multimodal understanding. The latency penalty is minimized by Gemini Flash's optimized inference speed, often faster than local Whisper + Llama 3 pipelines.
- **Output (Windows Native TTS):** The agent's response—which often contains highly sensitive confirmation details like "Transferring $5,000 to Account ending in 8899"—is

generated using the local Windows SAPI5 or WinRT engine. This ensures that the *confirmation of the transaction* never leaves the device's memory buffer. The synthesized audio is played directly to the user's speakers, creating an effective "Air Gap" for output data.

1.3 The "Click-to-Process" Interaction Model

Security in financial automation is not just about encryption; it is about intent verification. "Always-listening" agents (like Alexa or Siri) pose a theoretical risk of accidental activation. For a banking agent, an accidental activation could be catastrophic.

Arvyn v2.1 introduces a **Click-to-Start / Click-to-Process** input flow.

- **Click-to-Start:** The user presses and holds (or toggles) the activation button on the floating widget. This electrically engages the microphone and initializes the audio buffer.
- **Click-to-Process:** The user releases the button (or toggles off). This instantly seals the audio buffer, terminates the recording thread, and dispatches the payload to the Gemini Brain.

This physical interaction serves as a **"Proof of Presence,"** ensuring that no command is processed without explicit, tactile confirmation from the human operator. It creates a definitive "Session Boundary" for every transaction.

---

2. The Tech Stack: Architecture V2.1

The architecture is stratified into four distinct layers, each responsible for a specific aspect of the agent's cognition and actuation. This "Separation of Concerns" ensures that a failure in the GUI does not crash the banking session, and network latency in the brain does not freeze the interface.

2.1 The Presentation Layer: PyQt6 & The Omni-Widget

The interface must be persistent yet unobtrusive. We utilize **PyQt6** to create a custom, non-rectangular window.

- **Framework:** PyQt6 (v6.7.0).
- **Form Factor:** A floating, circular "Orb" or "Omni-Widget" that resides `AlwaysOnTop` of other windows.
- **Visual Logic:** The widget uses `QRegion.Ellipse` masking to achieve a perfect circle, breaking the traditional rectangular window paradigm.
- **Feedback Mechanism:**
    - **Idle:** Grey/Blue static ring.

    - **Recording (Click-to-Start):** Pulsing Red ring.
    - **Processing (Click-to-Process):** Spinning Amber loader.
    - **Execution:** Green flash upon successful Playwright action.
- **Threading:** The GUI runs on the main thread. All Audio I/O and Network calls are offloaded to `QThread` workers (e.g., `AudioWorker`, `BrainWorker`) to prevent "Application Not Responding" (ANR) events during heavy processing.

2.2 The Cognitive Layer: Gemini 1.5 Flash (Multimodal)

This is the most significant upgrade in v2.1. We replace the separate Speech-to-Text (STT) and LLM pipeline with a single **Multimodal Context Window**.

- **Engine:** Google Gemini 1.5 Flash via `google-genai` SDK.
- **Input:** Raw Audio Bytes (WAV/MP3) + Text (DOM Snapshot).
- **Advantage:** By feeding audio directly to the model, we preserve paralinguistic cues. A user hesitating ("Pay... 50 dollars?") is interpreted differently than a user speaking confidently ("Pay 50 dollars!"). The model can use this acoustic data to ask for confirmation if the confidence seems low.
- **Grounding:** The model is not just asked to "Pay the bill." It is fed a minified text representation of the current banking page's HTML (the DOM). It is instructed to map the user's voice command to the *specific* selectors present in that HTML, grounding its hallucinations in the reality of the current page.

2.3 The Kinetic Layer: Microsoft Playwright (Python)

The "Digital Hand" that manipulates the browser.

- **Driver:** Microsoft Playwright (v1.44.0).
- **Mode:** `headless=False`. Transparency is a security feature; the user must see the agent driving the browser to trust it.
- **Context Management:** We utilize persistent `BrowserContext` storage (`storage_state.json`) to save session cookies. This prevents the agent from needing to perform a full 2FA login on every single run, provided the bank's session token hasn't expired.
- **Targeting Strategy:** The agent primarily uses **Semantic Locators** (`get_by_role`, `get_by_label`) as identified by the Cognitive Layer, falling back to CSS selectors only when semantic tags are missing.

2.4 The Auditory Layer: Windows Native & PyAudio

The input/output audio interface.

- **Input:** `PyAudio` (v0.2.14). Captures raw PCM data from the default system microphone into a `BytesIO` memory buffer.
- **Output:** `pyttsx3` or `winsdk`. Interfaces with the Windows **Speech API (SAPI5)** or

Windows Runtime (WinRT). This ensures that the agent's voice is generated on-device, with zero latency and zero data leakage.

---

3. Deep Dive: The Cognitive Core (Gemini Integration)

The heart of Arvyn v2.1 is the `GeminiBrain` class. This component is responsible for orchestrating the interaction between the raw sensory data (Audio) and the structured environment (DOM).

3.1 Multimodal Ingestion Strategy

In traditional systems (v1.0), audio is first transcribed to text. This transcription is "lossy"—it strips away tone, pitch, and speed. Arvyn v2.1 sends the audio *blob* directly to Gemini.

3.1.1 The API Payload

Using the `google-genai` Python SDK, the payload is constructed as a multi-part content block.

1. **System Instruction:** Defines the persona ("You are a banking automation agent") and the output format (Strict JSON).
2. **Context Part:** A text string containing the cleaned, minified HTML of the current page. This tells the model *what is possible* on the screen.
3. **Audio Part:** The byte stream of the user's voice command.

3.1.2 Code Implementation Strategy

The implementation logic utilizes the `generate_content` method rather than the `chat` method for individual action steps. This is because each step in a browser automation flow introduces a radically different context (New Page = New DOM). Treating each step as a discrete "State Analysis" task is more robust than maintaining a chat history that might contain stale selectors from previous pages.

The code must handle the binary read of the audio buffer and formatting it correctly for the Gemini endpoint. The prompt engineering is crucial here: we explicitly instruct Gemini to return *only* JSON, with fields for `intent`, `selector`, `value`, and `reasoning`.

3.2 DOM Sanitization & Token Optimization

Sending the entire HTML of a modern banking dashboard to an LLM can be expensive and slow (high token count). Arvyn v2.1 implements a **DOM Sanitizer** before the data reaches Gemini.

- **Tag Stripping:** All `<script>`, `<style>`, `<svg>`, `<meta>`, and `<link>` tags are removed. These contain no semantic value for the agent's action logic.
- **Attribute Cleaning:** Attributes like `style="..."`, `onmouseover="..."`, and long base64

image strings in `src` attributes are stripped.
- **PII Redaction:** A RegEx pass runs over the text nodes to identify and redact patterns that look like Account Numbers (10-16 digits) or Balances. This ensures that while Gemini sees the *field* ("Account Number Input"), it does not see the *user's actual account number* if it happens to be displayed in a label.

---

4. Deep Dive: The Auditory Interface

4.1 The "Click-to-Process" Logic

This mechanism is the primary safety valve of the system. It is implemented via the `QThread` event loop in PyQt6.

1. **State: IDLE.** The `AudioRecorder` thread is instantiated but sleeping. The audio stream is closed.
2. **Signal: PRESSED.** When the user clicks the widget:
    - The Main Thread emits a `start_recording` signal.
    - The `AudioRecorder` opens the PyAudio stream (`paInt16`, 16000Hz, Mono).
    - It begins reading chunks (e.g., 1024 frames) from the hardware buffer and appending them to a localized list `self.frames`.
    - **Visual:** The Widget paints a pulsing red overlay.
3. **Signal: RELEASED.** When the user releases the click:
    - The stream is stopped and closed immediately.
    - The list `self.frames` is consolidated into a single `bytes` object using the `wave` library to add the correct RIFF headers.
    - This `audio_bytes` object is passed via signal to the `BrainWorker` thread.
    - **Visual:** The Widget paints an amber "Processing" spinner.

This logic ensures that the microphone is physically incapable of recording user conversation when the button is not depressed, effectively mitigating the "Hot Mic" risk.

4.2 Windows Native Text-to-Speech (TTS)

For output, we prioritize privacy. The feedback loop ("I have paid the bill") must not pass through a cloud server.

We utilize the **Microsoft Speech API (SAPI5)** via the `pyttsx3` library. This library provides a synchronous interface to the OS's TTS engine.

- **Voice Selection:** The initialization logic iterates through `engine.getProperty('voices')`. It specifically looks for voices with "Neural" or "Natural" in their metadata (common in Windows 11) to ensure a less robotic output. If unavailable, it falls back to the standard

"Microsoft David" or "Microsoft Zira" desktop voices.
- **Rate & Volume:** Financial information needs to be spoken clearly. We programmatically set the rate to 150 (slightly slower than conversational) to ensure digits in amounts ("5, 0, 0, 0") are distinct.

---

5. Deep Dive: The Kinetic Layer (Targeting Rio Finance)

The prompt specifies targeting rio_finance_bank. Based on research into the developer's (Roshan-Choudhary) repositories, we can infer the architectural patterns of this application. It is likely a **Spring Boot** backend with a **React** or **Thymeleaf** frontend.

5.1 Handling The "Rio" Application Structure

Applications in this portfolio typically follow a standard structure:

1. **Landing/Login:** A clean page with a central form card.
2. **Dashboard:** A grid layout displaying "Account Balance," "Recent Transactions," and a Sidebar for navigation.
3. **Bill Pay/Transfer:** Modal dialogs or dedicated form pages.

5.2 The "Blind" Selector Strategy

Since the live URL is currently inaccessible (404), Arvyn v2.1 employs a **Heuristic Selector Strategy**. We do not hardcode rigid XPaths (e.g., /div/input). Instead, we construct a hierarchy of lookup strategies:

Strategy A: Semantic ID/Name Match

Gemini is instructed to look for IDs that semantically match the intent.

- *Intent:* Login -> *Target:* #login, #signin, #user-auth.
- *Intent:* Password -> *Target:* #password, #pass, #pwd.

Strategy B: Accessibility Roles (ARIA)

If IDs are obfuscated (common in React builds), we use Playwright's role-based locators.

- page.get_by_role("button", name="Login")
- page.get_by_placeholder("Enter Account Number")

Strategy C: Text Anchoring

As a final fallback, we use text coupling.

- Find the label containing text "Electricity Provider".
- Find the input element associated with that label ID.

This tiered strategy ensures that Arvyn v2.1 is resilient to minor code changes in the Rio Finance application. Even if the developer changes the ID of the login button from `btn-login` to `btn-submit`, the **Strategy B** (Role: Button, Name: Login) will still succeed.

5.3 Authentication & Session Persistence

Banking portals often have short session timeouts. Arvyn handles this via a `SessionManager` class.

- **On Launch:** It attempts to inject the `storage_state.json` (cookies) into the browser context.
- **Validation:** It navigates to the Dashboard URL. If the URL redirects to `/login`, the agent knows the session is dead.
- **Re-Authentication:** It triggers a "Login Interrupt." The Voice interface speaks: "Session expired. Please authorize." The user can then manually enter the password (for security) or allow the agent to autofill from a secure, encrypted local vault.

---

6. Implementation Blueprint

This section provides the complete, expanded file structure required for a production-grade application.

6.1 Comprehensive Directory Structure

The previous "flat" structure has been expanded to support scalability, asset management, and strict separation of concerns (SoC).

```
Arvyn_V2.1_Enterprise/
├── main.py # Application entry point, initializes AppContext
├── config.py # [Config] Global constants, timeouts, and API keys
├── requirements.txt # Python dependency manifest
├──.env # Environment variables (Gemini API Key, Bank URLs)
│
├── assets/ # Static assets for the UI
│ ├── icons/
│ │ ├── mic_active.svg # Vector icon for listening state
│ │ ├── mic_idle.svg # Vector icon for idle state
```

```
│ │ └── loader.gif # Processing animation
│ └── styles/
│ └── theme.qss # PyQt6 stylesheet (CSS-like) for the Widget
│
├── core/ # [Cognition] The 'Brain' logic
│ ├── init.py
│ ├── brain_engine.py # Gemini 1.5 Flash client (Multimodal request
handling)
│ ├── intent_parser.py # JSON validation and fallback logic
│ └── secure_store.py # Encrypted local vault for user credentials (using
'cryptography' lib)
│
├── kinetic/ # [Actuation] The 'Hands' logic (Playwright)
│ ├── init.py
│ ├── navigator.py # Main Playwright controller (Browser launch, page
routing)
│ ├── actions.py # Specific atomic actions (Click, Fill, Read, Wait)
│ ├── dom_sanitizer.py # HTML cleaning to reduce token usage before
sending to Gemini
│ └── auth_handler.py # Special logic for handling Login/OTP flows
│
├── io_modules/ # Input/Output hardware interfaces
│ ├── init.py
│ ├── audio_recorder.py # PyAudio low-level stream handler
(Click-to-Process logic)
│ └── voice_synthesizer.py # Windows Native TTS (pyttsx3/SAPI5 wrapper)
│
├── gui/ # [Presentation] All Visual Components
│ ├── init.py
│ ├── app_context.py # Global state manager for the GUI
│ ├── signals.py # Centralized definition of all PyQT Signals
│ ├── animations.py # Logic for the "Expand" and "Collapse" window
transitions
│ ├── components/
│ │ ├── orb_window.py # The circular floating widget (Draggable)
│ │ └── dashboard.py # The expanded rectangular Command Center
│ └── workers.py # QThread classes to keep heavy logic off the UI thread
│
└── data/ # [Persistence] Local state storage
├── session_state.json # Browser cookies (Playwright storage state)
└── logs/
└── activity.log # Rolling log file for debugging
```

6.2 Key File Responsibilities

6.2.1 `gui/components/orb_window.py` (The Circular Popup)

This file is responsible specifically for the "Resting State" of the agent.

- **Draggable Logic:** Overrides `mousePressEvent` and `mouseMoveEvent` to calculate delta vectors, allowing the frameless window to be dragged around the screen.
- **Masking:** Applies `setMask(QRegion(..., QRegion.Ellipse))` to ensure the window is perfectly circular, not just a square with transparent corners.
- **Expansion Logic:** Listens for a click event. On click, it emits a signal to `main.py` to trigger the transformation to the Dashboard.

6.2.2 `gui/components/dashboard.py` (The Command Center)

This file defines the expanded UI.

- **Layout:** A `QVBoxLayout` containing the Interaction Log (QTextEdit, read-only) and the Command Bar (QLineEdit + Mic Button).
- **Action Buttons:** Hidden by default. If the Brain returns a `CONFIRMATION_REQUIRED` intent, this view dynamically renders "Approve" and "Disapprove" buttons.

6.2.3 `core/brain_engine.py` (Gemini Interface)

Updated to handle the multimodal payload.

Python

```
import os

import google.generativeai as genai

from dotenv import load_dotenv


load_dotenv()


class GeminiBrain:

    def __init__(self):

        genai.configure(api_key=os.getenv("GEMINI_API_KEY"))

        # Using the Flash model for low-latency
```

```
        self.model = genai.GenerativeModel('gemini-1.5-flash')

    def analyze_intent(self, audio_bytes, dom_context):
        """
        Sends audio + DOM to Gemini and retrieves JSON intent.
        """
        prompt = """
        You are an autonomous agent interacting with a banking portal.
        Analyze the audio command and the provided HTML DOM.
        Return a JSON object with these keys:
        - "intent": The user's goal (LOGIN, NAVIGATE, FILL, READ).
        - "selector": The best CSS selector to interact with.
        - "value": The value to type (if applicable).
        - "speech": A short confirmation phrase to speak to the user.
        """

        # Construct the multipart payload
        # Note: In production, audio_bytes might need wrapping in a Blob object
        # depending on specific SDK version requirements.
        response = self.model.generate_content([prompt, dom_context, {"mime_type": "audio/wav",
"data": audio_bytes}])

        try:
            return self._extract_json(response.text)
        except Exception as e:
```

```
        return {"intent": "ERROR", "speech": "I could not process that command."}

    def _extract_json(self, text):
        # Implementation to strip markdown code blocks and parse JSON
        import json
        text = text.replace("```json", "").replace("```", "").strip()
        return json.loads(text)
```

6.2.4 gui/animations.py (Visual Polish)

This module handles the smooth transition between the circular Orb and the rectangular Dashboard.

- Uses `QPropertyAnimation` on the `geometry` property of the main window.
- **Expand:** Animates size from `(100, 100)` to `(400, 600)` and radius from `50` to `10`.
- **Collapse:** Reverses the animation.

6.3 Technical Data Tables

Component	V2.0 (Old)	V2.1 (New)	Reasoning
Logic Brain	Local SLM / Regex	Gemini 1.5 Flash	massive context window, native multimodal reasoning.
Speech Input	Local Whisper (Pipeline)	Gemini Direct Ingestion	Reduces latency by combining STT+Logic; preserves tone.
Speech Output	Google Cloud TTS	Windows Native (SAPI5)	**Zero Trust:** Sensitive financial feedback never leaves the device.

Trigger	Wake Word ("Hey Arvyn")	Click-to-Process	**Security:** Eliminates "Hot Mic" vulnerability; confirms intent.
Targeting	Hardcoded XPaths	Semantic Anchoring	Resilience against UI updates in the `rio_finance_bank` portal.

---

7. Operational Workflow: The "Life of a Command"

To illustrate the system in action, we trace the lifecycle of a specific command: **"Transfer 500 dollars to the electricity bill."**

1. **Initiation (The Handshake):** The user physically presses and holds the center of the Arvyn Widget. The UI glows Red.
2. **Capture (The Buffer):** The `AudioRecorder` captures the phrase "Transfer 500 dollars..." into a memory buffer.
3. **Transmission (The Handoff):** The user releases the button. The Widget spins Amber. The audio blob is bundled with the current DOM of the Rio Finance Dashboard and sent to Gemini.
4. **Cognition (The Brain):**
    - Gemini 1.5 Flash receives the audio and the DOM.
    - It "hears" the intent: `PAY_BILL`.
    - It scans the DOM and finds a button with `id="nav-utility-pay"`.
    - It returns JSON: `{"intent": "CLICK", "selector": "#nav-utility-pay", "speech": "Opening utility payments."}`.
5. **Actuation (The Hand):**
    - The `BrowserAgent` receives the JSON.
    - It executes `page.click("#nav-utility-pay")`.
    - The browser navigates to the payment form.
6. **Feedback (The Voice):**
    - Simultaneously, the `WindowsSpeaker` receives the text "Opening utility payments."
    - It synthesizes the speech locally using the SAPI5 engine.
7. **Completion:** The browser is now on the Bill Pay page, ready for the next command (e.g., "Select Provider").

---

8. Security Protocols and Governance

8.1 Data Sanitization Pipeline

Arvyn employs a rigorous sanitization pipeline before any data leaves the local machine.

- **RegEx Filtering:** All text nodes in the DOM are scanned for sequences matching Credit Card numbers (Luhn algorithm check) or high-entropy strings (Session Tokens). These are replaced with `` tags.
- **Image Stripping:** All `<img>` tags are stripped to prevent the inadvertent transmission of QR codes or sensitive visual data to the Gemini API.

8.2 The "Air-Gapped" Output

The choice of Windows Native TTS is non-negotiable for security. By generating speech locally, Arvyn ensures that the specific confirmation details of a transaction (amounts, account names) are never processed by a third-party server. This creates a unidirectional data flow for sensitive output: **Browser RAM -> Local TTS -> Speaker**. No network packets are involved.

8.3 Error Boundaries & Fail-Safes

- **Confidence Thresholds:** If Gemini returns an action with low confidence (or ambiguous reasoning), Arvyn enters a **"Clarification Mode."** It does *not* click. Instead, it speaks: "I am not sure which account you mean. Please clarify."
- **The "Conscious Pause":** For any action classified as `CRITICAL` (e.g., clicking "Confirm Transfer"), Arvyn triggers a UI Interrupt. The Widget expands to show a "CONFIRM?" button. The user must manually click this button to proceed. This effectively keeps a "Human-in-the-Loop" for all money-movement actions.

---

9. Conclusion

Agent Arvyn v2.1 defines a new standard for personal financial automation. By synthesizing the raw, multimodal power of **Gemini 1.5 Flash** with the surgical precision of **Playwright**, and wrapping it in a privacy-first, **Windows Native** shell, we have created an agent that is both powerful and trustworthy.

This architecture successfully addresses the unique challenges of the **Rio Finance Bank** target environment—handling SPA navigation, session management, and dynamic DOMs—without compromising user security. The shift to a **Click-to-Process** interaction model further reinforces this security posture, ensuring that Arvyn remains a tool of explicit intent, never a passive observer. This blueprint provides the complete, validated roadmap for

the immediate development of the v2.1 prototype.