Clone Detection System — Plan & Starter Kit

Below is a complete, **from-scratch plan** and a **ready-to-run starter implementation** that uses:

- **Google Cloud Vision API** (service account JSON) for **logo + OCR** signals.
- **Gemini (Multimodal)** using **GEMINI_API_KEY** for **visual+text reasoning** and a natural-language explanation.
- **URL & HTML heuristics** for classic phishing indicators.
- A small **Flask** API that orchestrates everything.

You can run only Vision, only Gemini, or both together.

---

1) What we're building (high level)

Goal: Given a **URL** and/or a **screenshot**, decide whether the page is a **clone** (phish) of an authorized site. Return a **score**, **decision**, and a **clear explanation**.

Signals we combine

1. **Visual (Gemini multimodal):**
2. Provide the screenshot + compact context (URL, title, extracted text snippets).
3. Ask Gemini: *"Does this look like a clone of any authorized site? Why?"*
4. Return a **likelihood score** + **explanation**.
5. **Logo & Text (Vision API):**
6. **LOGO_DETECTION**: Find brand logos.
7. **OCR**: Extract brand names/keywords.
8. If a brand is detected but **domain doesn't match** the authorized domains: raise risk.
9. **URL/HTML Heuristics:**
10. Suspicious host patterns (punycode `xn--`, IP-in-host, many hyphens, long subdomains).
11. Mismatch between **visible brand** vs **registered domain**.
12. Presence of login keywords, credential forms.

Decisioning

- We compute a **final score (0–100)** = weighted blend of:
- Gemini judgement (0–100) → **40%**
- Logo/Brand mismatch (0–100) → **40%**
- URL/HTML heuristics (0–100) → **20%**
- **Decision:**

- `clean` if score < 30
- `suspicious` if 30 ≤ score < 60
- `clone` if score ≥ 60

You can tweak weights & thresholds in `config/app.yaml`.

---

2) Project layout

```
clone-detector/
├─ .env.example
├─ requirements.txt
├─ config/
│  ├─ app.yaml                 # thresholds, weights
│  └─ authorized_sites.yaml    # your allowed brands & canonical domains
├─ reference/
│  └─ logos/                   # optional: known logos for offline checks
├─ app.py                      # Flask entrypoint
├─ detectors/
│  ├─ gemini_mm.py             # Gemini multimodal reasoning
│  ├─ vision_signals.py        # Vision API (logo/ocr)
│  └─ heuristics.py            # URL/HTML features & scoring
├─ utils/
│  ├─ screenshot.py            # Playwright page fetch & screenshot
│  ├─ net.py                   # URL validation, SSRF guard
│  └─ text.py                  # text cleanup helpers
└─ config_loader.py            # load yaml configs safely
```

---

3) Prerequisites

- **Python 3.10+**
- **Google Cloud** project with **Vision API** enabled, plus a **service account JSON** file.
- **Gemini API key** (for `google-generativeai`).
- (Optional) Your curated **authorized sites** in `config/authorized_sites.yaml`.

---

4) Environment setup (virtualenv)

macOS / Linux

```
python3 -m venv .venv
source .venv/bin/activate
```

```
python -m pip install --upgrade pip
pip install -r requirements.txt
# Install Playwright browser
python -m playwright install chromium
```

Windows (PowerShell)

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
pip install -r requirements.txt
# Install Playwright browser
python -m playwright install chromium
```

.env

Copy `.env.example` ⇒ `.env` and set:

```
GEMINI_API_KEY=your_gemini_key_here
GOOGLE_APPLICATION_CREDENTIALS=abs_path_to_your_service_account.json
PROJECT_ID=your-gcp-project-id
LOCATION=us  # or your region for Vertex features you may add later
```

> Note: `GOOGLE_APPLICATION_CREDENTIALS` must be an **absolute path**.

---

5) `requirements.txt`

```
Flask==3.0.3
google-generativeai==0.7.2
google-cloud-vision==3.7.4
playwright==1.46.0
tldextract==5.1.2
python-dotenv==1.0.1
beautifulsoup4==4.12.3
pydantic==2.8.2
PyYAML==6.0.2
idna==3.7
```

> Pinned versions for reproducibility. You can relax later.

---

6) `config/app.yaml`

```yaml
weights:
  gemini: 0.4
  vision_brand: 0.4
  heuristics: 0.2

thresholds:
  clone: 60
  suspicious: 30

limits:
  max_html_chars: 20000
  screenshot_timeout_ms: 15000
  nav_timeout_ms: 15000
```

---

7) `config/authorized_sites.yaml`

```yaml
brands:
  - name: Google
    domains: ["google.com", "accounts.google.com"]
    keywords: ["Google", "Gmail", "Sign in"]
  - name: PayPal
    domains: ["paypal.com"]
    keywords: ["PayPal", "Log in", "Send & Request"]
  - name: Microsoft
    domains: ["microsoft.com", "login.live.com"]
    keywords: ["Microsoft", "Outlook", "Sign in"]
```

Add your own brands + canonical domains.

---

8) `config_loader.py`

```python
from __future__ import annotations
import yaml
from pathlib import Path

class AppConfig:
    def __init__(self, root: Path):
        self.root = root
        self.app = self._load_yaml(root / "config" / "app.yaml")
```

```
        self.auth = self._load_yaml(root / "config" / "authorized_sites.yaml")

    def _load_yaml(self, p: Path):
        with open(p, "r", encoding="utf-8") as f:
            return yaml.safe_load(f)

    @property
    def weights(self):
        return self.app.get("weights", {})

    @property
    def thresholds(self):
        return self.app.get("thresholds", {})

    @property
    def limits(self):
        return self.app.get("limits", {})

    @property
    def brands(self):
        return self.auth.get("brands", [])
```

---

9) `utils/net.py`

```
from __future__ import annotations
import ipaddress, socket, re
from urllib.parse import urlparse

PRIVATE_NETS = [
    ipaddress.ip_network("10.0.0.0/8"),
    ipaddress.ip_network("172.16.0.0/12"),
    ipaddress.ip_network("192.168.0.0/16"),
    ipaddress.ip_network("127.0.0.0/8"),
    ipaddress.ip_network("169.254.0.0/16"),
    ipaddress.ip_network("::1/128"),
]

HOST_REGEX = re.compile(r"^[A-Za-z0-9.-]+$")

class UrlError(ValueError):
    pass

def is_private_ip(ip: str) -> bool:
    try:
```

```
        ipa = ipaddress.ip_address(ip)
    except ValueError:
        return False
    return any(ipa in net for net in PRIVATE_NETS)

def guard_url(url: str) -> str:
    """Validate and block SSRF targets (only http/https, no private nets)."""
    p = urlparse(url)
    if p.scheme not in {"http", "https"}:
        raise UrlError("Only http/https allowed")
    if not p.hostname or not HOST_REGEX.match(p.hostname):
        raise UrlError("Invalid hostname")
    # Resolve and block private IPs
    try:
        infos = socket.getaddrinfo(p.hostname, None)
        for info in infos:
            ip = info[4][0]
            if is_private_ip(ip):
                raise UrlError("Target resolves to private/loopback address")
    except socket.gaierror:
        raise UrlError("DNS resolution failed")
    return url
```

---

10) `utils/screenshot.py`

```
from __future__ import annotations
import asyncio
from pathlib import Path
from playwright.async_api import async_playwright

async def _shot(url: str, out_path: Path, nav_timeout_ms=15000):
    async with async_playwright() as p:
        browser = await p.chromium.launch()
        context = await browser.new_context(viewport={"width": 1366, "height":
768})
        page = await context.new_page()
        page.set_default_navigation_timeout(nav_timeout_ms)
        await page.goto(url)
        await page.screenshot(path=str(out_path), full_page=True)
        title = await page.title()
        html = await page.content()
        await browser.close()
        return title, html
```

```
def take_screenshot(url: str, out_path: Path, nav_timeout_ms=15000):
    return asyncio.run(_shot(url, out_path, nav_timeout_ms))
```

---

11) `utils/text.py`

```
import re
from bs4 import BeautifulSoup

LOGIN_WORDS = {"login", "log in", "sign in", "password", "account", "verify"}

def extract_text_snippet(html: str, limit=2000):
    soup = BeautifulSoup(html, "html.parser")
    text = soup.get_text(" ", strip=True)
    text = re.sub(r"\s+", " ", text)
    return text[:limit]

def contains_login_words(text: str) -> bool:
    low = text.lower()
    return any(w in low for w in LOGIN_WORDS)
```

---

12) `detectors/heuristics.py`

```
from __future__ import annotations
import re
import tldextract
from urllib.parse import urlparse
from utils.text import contains_login_words

PUNYCODE = "xn--"

class HeuristicResult(dict):
    pass

def url_risk(url: str, page_text: str = "") -> HeuristicResult:
    p = urlparse(url)
    host = p.hostname or ""
    ext = tldextract.extract(url)
    registered = ".".join([ext.domain, ext.suffix]) if ext.suffix else
ext.domain
    subdomain = ext.subdomain or ""
```

```
    risk = 0
    signals = {}

    # Suspicious patterns
    if PUNYCODE in host:
        risk += 20; signals["punycode"] = True
    if re.search(r"\d+\.\d+\.\d+\.\d+", host):
        risk += 25; signals["ip_in_host"] = True
    if host.count("-") >= 3:
        risk += 10; signals["many_hyphens"] = True
    if len(subdomain) > 30:
        risk += 10; signals["long_subdomain"] = True
    if contains_login_words(page_text):
        risk += 10; signals["login_words"] = True

    return HeuristicResult({
        "risk": min(risk, 100),
        "host": host,
        "registered_domain": registered,
        "subdomain": subdomain,
        "signals": signals,
    })


def brand_mismatch_score(brand: str, detected_brand: str, domain: str,
allowed_domains: list[str]) -> int:
    if not detected_brand:
        return 0
    if domain in allowed_domains:
        return 0
    # If we see the brand but domain doesn't match canonical -> high risk
    return 80
```

---

13) detectors/vision_signals.py

```
from __future__ import annotations
import io
from typing import Optional
from google.cloud import vision

class VisionFindings(dict):
    pass

_client = None
```

```
def _client_once():
    global _client
    if _client is None:
        _client = vision.ImageAnnotatorClient()
    return _client


def analyze_image(image_bytes: bytes) -> VisionFindings:
    client = _client_once()
    image = vision.Image(content=image_bytes)

    logos = client.logo_detection(image=image).logo_annotations
    ocr = client.document_text_detection(image=image)

    found_logos = [
        {"description": l.description, "score": l.score}
        for l in logos if l.description
    ]
    text = ocr.full_text_annotation.text if ocr.full_text_annotation else ""

    return VisionFindings({
        "logos": found_logos,
        "text": text,
    })
```

---

14) `detectors/gemini_mm.py`

```
from __future__ import annotations
import os, base64
import google.generativeai as genai

_model = None

def _model_once():
    global _model
    if _model is None:
        api_key = os.environ.get("GEMINI_API_KEY")
        if not api_key:
            raise RuntimeError("GEMINI_API_KEY not set")
        genai.configure(api_key=api_key)
        _model = genai.GenerativeModel("gemini-1.5-pro")
    return _model
```

```
PROMPT = (
    "You are a security analyst. Given a webpage screenshot and context, "
    "judge whether it is likely a clone (phishing) of any authorized site.\n"
    "Return a JSON with keys: likelihood (0-100), suspected_brand (string or
''), "
    "explanation (1-3 sentences). "
    "Use layout, logo/iconography, color scheme, typography, and UI patterns. "
    "Be conservative; only high similarity implies high likelihood."
)


def judge_with_image(image_bytes: bytes, url: str, page_title: str,
text_snippet: str, authorized_brands: list[dict]):
    model = _model_once()
    auth_desc = ", ".join(
        f"{b['name']} (domains: {', '.join(b.get('domains', []))})" for b in
authorized_brands
    ) or "(none provided)"

    parts = [
        {
            "text": f"URL: {url}\nTitle: {page_title}\nAuthorized brands:
{auth_desc}\nText snippet: {text_snippet[:1200]}"
        },
        {"mime_type": "image/png", "data": image_bytes},
        {"text": PROMPT},
    ]

    resp = model.generate_content(parts)
    out = resp.text or "{}"
    # Try forgiving parse
    import json
    try:
        data = json.loads(out)
    except Exception:
        # fallback: extract numbers and guess
        data = {"likelihood": 50, "suspected_brand": "", "explanation": out[:
300]}
    # Clamp
    data["likelihood"] = max(0, min(100, int(data.get("likelihood", 50))))
    data["suspected_brand"] = data.get("suspected_brand", "")
    data["explanation"] = data.get("explanation", "")
    return data
```

---

15) `app.py` (Flask API)

```python
from __future__ import annotations
import os, io, json
from pathlib import Path
from flask import Flask, request, jsonify
from dotenv import load_dotenv
from config_loader import AppConfig
from utils.net import guard_url, UrlError
from utils.screenshot import take_screenshot
from utils.text import extract_text_snippet
from detectors.heuristics import url_risk, brand_mismatch_score
from detectors.vision_signals import analyze_image
from detectors.gemini_mm import judge_with_image

ROOT = Path(__file__).resolve().parent
load_dotenv(ROOT / ".env")

app = Flask(__name__)
CFG = AppConfig(ROOT)

OUT_DIR = ROOT / "_artifacts"; OUT_DIR.mkdir(exist_ok=True)

@app.post("/analyze")
def analyze():
    """
    Accepts either:
    - JSON: {"url": "https://..."}
    - multipart/form-data: fields: url (optional), screenshot (file)
    Returns a JSON verdict.
    """
    url = request.form.get("url") or (request.json or {}).get("url")

    image_bytes = None
    if "screenshot" in (request.files or {}):
        image_bytes = request.files["screenshot"].read()

    title = ""; html = ""; text_snippet = ""

    if url:
        try:
            guard_url(url)
        except UrlError as e:
            return jsonify({"error": f"URL blocked: {e}"}), 400

    # If no screenshot was uploaded, try to capture one
```

```
    if not image_bytes and url:
        shot_path = OUT_DIR / "shot.png"
        try:
            title, html = take_screenshot(url, shot_path,
CFG.limits.get("nav_timeout_ms", 15000))
            image_bytes = shot_path.read_bytes()
        except Exception as e:
            return jsonify({"error": f"Failed to fetch page: {e}"}), 502

    if html:
        text_snippet = extract_text_snippet(html, limit=3000)

    # Heuristics (URL-based)
    heur = url_risk(url or "", page_text=text_snippet)

    # Vision signals
    vision = {"logos": [], "text": ""}
    if image_bytes:
        try:
            vision = analyze_image(image_bytes)
        except Exception as e:
            vision = {"error": f"vision_failed: {e}", "logos": [], "text": ""}

    # Map vision logos to an authorized brand (best-effort)
    detected_brand = ""
    if vision.get("logos"):
        # pick top logo description by score
        logos = sorted(vision["logos"], key=lambda x: x.get("score", 0),
reverse=True)
        detected_brand = logos[0]["description"]

    # Gemini judgement
    gem = {"likelihood": 50, "suspected_brand": "", "explanation": ""}
    if image_bytes:
        try:
            gem = judge_with_image(
                image_bytes=image_bytes,
                url=url or "",
                page_title=title,
                text_snippet=text_snippet,
                authorized_brands=CFG.brands,
            )
        except Exception as e:
            gem = {"likelihood": 50, "suspected_brand": "", "explanation":
f"gemini_failed: {e}"}

    # Compute brand mismatch score
    allowed_domains = []
```

```
    brand_for_mismatch = None
    reg_domain = heur.get("registered_domain", "")

    # If Gemini guessed a brand, prefer that; else use Vision top logo
    if gem.get("suspected_brand"):
        brand_for_mismatch = gem["suspected_brand"]
    elif detected_brand:
        brand_for_mismatch = detected_brand

    if brand_for_mismatch:
        for b in CFG.brands:
            if b["name"].lower() in brand_for_mismatch.lower():
                allowed_domains = b.get("domains", [])
                break

    brand_score = brand_mismatch_score(brand_for_mismatch or "",
brand_for_mismatch or "", reg_domain, allowed_domains)

    # Blend scores
    w = CFG.weights
    final = (
        w.get("gemini", 0.4) * gem.get("likelihood", 50) +
        w.get("vision_brand", 0.4) * brand_score +
        w.get("heuristics", 0.2) * heur.get("risk", 0)
    )

    # Decision
    th = CFG.thresholds
    if final >= th.get("clone", 60):
        decision = "clone"
    elif final >= th.get("suspicious", 30):
        decision = "suspicious"
    else:
        decision = "clean"

    return jsonify({
        "url": url,
        "decision": decision,
        "score": round(final, 1),
        "signals": {
            "heuristics": heur,
            "vision": {k: v for k, v in vision.items() if k != "error"},
            "gemini": gem,
            "brand_mismatch": {
                "brand": brand_for_mismatch,
                "allowed_domains": allowed_domains,
                "registered_domain": reg_domain,
                "score": brand_score,
```

```
            },
        },
        "explanation": (
            gem.get("explanation") or
            (f"Detected brand {brand_for_mismatch} on non-canonical domain
{reg_domain}." if brand_score else "No strong clone indicators.")
        )[:600],
        "errors": {k: v for k, v in {"vision": vision.get("error")}.items() if
v}
    })

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int(os.environ.get("PORT", 5000)), debug=True)
```

---

16) `.env.example`

```
GEMINI_API_KEY=
GOOGLE_APPLICATION_CREDENTIALS=/absolute/path/to/service-account.json
PROJECT_ID=
LOCATION=us
```

---

17) Run & test

Start the API:

```
flask --app app run --port 5000 --debug
```

Send a URL (server auto-screenshots):

```
curl -X POST http://localhost:5000/analyze
  -H 'Content-Type: application/json'
  -d '{"url":"https://example.com"}'
```

Or upload your own screenshot:

```
curl -X POST http://localhost:5000/analyze
  -F url=https://suspicious.example
  -F screenshot=@/path/shot.png
```

Sample response (shape):


```
{
  "url": "https://...",
  "decision": "suspicious",
  "score": 62.3,
  "signals": {
    "heuristics": {"risk": 30, "registered_domain": "paypa1-login.com", ...},
    "vision": {"logos": [{"description":"PayPal","score":0.88}], "text": "..."},
    "gemini": {"likelihood": 70, "suspected_brand": "PayPal", "explanation":
"..."},
    "brand_mismatch": {"brand":"PayPal","allowed_domains":
["paypal.com"],"registered_domain":"paypa1-login.com","score":80}
  },
  "explanation": "Layout and logo match PayPal, but domain is not paypal.com.",
  "errors": {}
}
```


---

18) Why this design works (and its limits)

- **Defense in depth:** Vision + Gemini + heuristics reduce reliance on any single signal.
- **Explainability:** Gemini's natural-language reason + explicit brand/domain mismatch.
- **False positives control:** Conservative thresholds; you can ratchet up/down.
- **Privacy/Safety:** SSRF guard blocks internal targets.

Limits & next steps: - Some clones are **brand-agnostic** (generic login skins). Use more HTML/JS DOM cues. - Consider **Embedding similarity** (store reference screenshots and compute image embeddings; match via cosine similarity). You can later plug in Vertex AI Matching Engine. - Add **URL reputation** (Safe Browsing API) and **TLS cert issuer** check. - Stream logs to **BigQuery**; add **audit UI** to review cases.

---

19) Hardening toggles & knobs

- **Change weights/thresholds** in `config/app.yaml`.
- **Expand brand catalog** in `config/authorized_sites.yaml`.
- **Tighten SSRF guard** (block non-80/443 ports, etc.).
- **Time limits** for navigation/screenshot in `limits`.

---

20) Minimal prompt (Gemini)

```
You are a security analyst. Given one webpage screenshot and context, decide if
it likely clones any authorized site.
Return JSON: { likelihood: 0-100, suspected_brand: string, explanation:
string }.
Use layout, logos, color scheme, typography, and UI structure; be conservative.
```

---

21) Troubleshooting

- **Vision API auth error:** Ensure `GOOGLE_APPLICATION_CREDENTIALS` points to your service-account JSON and that **Vision API** is enabled.
- **Gemini auth error:** Ensure `GEMINI_API_KEY` is set. For Vertex Gemini instead, swap to the `vertexai` SDK.
- **Playwright timeouts:** The site may block automation; try higher timeouts or user-agent tweaks (add later to `screenshot.py`).
- **JSON parse issues from Gemini:** We already guard with a permissive fallback; you can also ask for `application/json` in the prompt.

---

22) Optional: switch to Vertex AI Gemini (service account)

Later, if you prefer **Vertex AI** instead of the API key route, you can replace `detectors/gemini_mm.py` with the Vertex SDK (`vertexai`), initialize with `project`, `location`, and use `GenerativeModel("gemini-1.5-pro-vision")`. The rest of the pipeline stays the same.

You're set. Start with sections 4–5 (env & deps), then add your service account JSON + Gemini key, and run the API. Adjust thresholds after a few real samples.