A minimal “beta” Python prototype can be initialized as a small, testable package with clear entry points, config, and room to grow. Below is a concrete skeleton you can drop into a new repo and start iterating on.

Project layout
text
device_lexicon_proto/
├─ device_lexicon/
│ ├─ **init**.py
│ ├─ config.py
│ ├─ ocr.py
│ ├─ parser.py
│ ├─ lexicon.py
│ └─ cli.py
├─ tests/
│ ├─ test_ocr.py
│ ├─ test_parser.py
│ └─ test_lexicon.py
├─ pyproject.toml
├─ README.md
└─ .gitignore

This structure gives you a Python package (device_lexicon) plus tests and packaging metadata, which is the current best practice for new projects.gist.github+1
Core beta modules
config.py
python
from dataclasses import dataclass
from pathlib import Path

@dataclass
class OCRConfig:
language: str = "eng"
psm: int = 6 # assume uniform text block

@dataclass
class ProjectPaths:

root: Path
images_dir: Path
output_dir: Path

```
@classmethod
def from_root(cls, root: Path) -> "ProjectPaths":
    return cls(
        root=root,
        images_dir=root / "images",
        output_dir=root / "output",
    )
```

ocr.py (placeholder abstraction; you can back it with Tesseract, EasyOCR, etc.)
python
from pathlib import Path
from typing import Dict
from .config import OCRConfig

def run_ocr_on_image(image_path: Path, cfg: OCRConfig) → str:
"""
Beta stub: plug in your preferred OCR engine here.
"""
TODO: integrate pytesseract or other library
raise NotImplementedError("OCR backend not wired up yet.")

def batch_ocr(image_paths: list[Path], cfg: OCRConfig) → Dict[str, str]:
return {p.name: run_ocr_on_image(p, cfg) for p in image_paths}

parser.py (Android-style property and log parsing based on AOSP logging format).
source.android+1
python
from dataclasses import dataclass
from typing import Optional

@dataclass
class PropEntry:
key: str
value: str
source_line: str

def parse_prop_line(line: str) → Optional[PropEntry]:
"""
Parse lines like 'ro.build.version.release=14' into key/value.
Returns None if it doesn't look like a property.
"""
if "=" not in line:
return None
key, value = line.split("=", 1)

key = key.strip()
value = value.strip()
if not key.startswith(("ro.", "dalvik.", "ro_boot.", "ro.boot.")):
return None
return PropEntry(key=key, value=value, source_line=line.rstrip())

lexicon.py
python
from collections import defaultdict
from dataclasses import dataclass, field
from typing import Dict, List
from .parser import parse_prop_line, PropEntry

@dataclass
class ImageLexicon:
image_name: str
properties: Dict[str, str] = field(default_factory=dict)
tokens: List[str] = field(default_factory=list)

def build_lexicon_from_text(image_name: str, text: str) -> ImageLexicon:
lex = ImageLexicon(image_name=image_name)
for line in text.splitlines():
# 1) property keys
prop = parse_prop_line(line)
if prop:
lex.properties[prop.key] = prop.value

```
    # 2) quick-and-dirty tokenization for other markers
    for raw_tok in line.split():
        tok = raw_tok.strip("[]:,()")
        if any(tok.startswith(pfx) for pfx in ("ro.", "dalvik.", "ro.boot.", "ro.vendor.'
            continue  # already captured as property
        if tok and tok not in lex.tokens:
            lex.tokens.append(tok)
return lex
```

cli.py
python
import json
from pathlib import Path
from .config import OCRConfig, ProjectPaths
from .ocr import batch_ocr
from .lexicon import build_lexicon_from_text

def main():
root = Path(".").resolve()
paths = ProjectPaths.from_root(root)
paths.output_dir.mkdir(parents=True, exist_ok=True)

```
images = sorted(paths.images_dir.glob("*.jpg"))
ocr_cfg = OCRConfig()

# NOTE: wire in real OCR before use
ocr_results = batch_ocr(images, ocr_cfg)

all_lexicons = {}
for name, text in ocr_results.items():
    lex = build_lexicon_from_text(name, text)
    all_lexicons[name] = {
        "properties": lex.properties,
        "tokens": lex.tokens,
    }

with open(paths.output_dir / "lexicons.json", "w", encoding="utf-8") as f:
    json.dump(all_lexicons, f, indent=2)
```

if **name** == "**main**":
main()

Getting this running
Create the folder structure and files above in a new git repo.
Add a simple pyproject.toml using Poetry or bare setuptools so the package is installable.
gist.github
Drop your recovery-photo JPGs into images/, implement run_ocr_on_image() against Tesseract or another OCR engine, then run python -m device_lexicon.cli to emit output/lexicons.json.
This gives you a concrete beta prototype wired around the photographed recovery logs, and you can iteratively refine parsers, lexicon schema, and downstream analysis from here.
https://gist.github.com/Pulimet/5013acf2cd5b28e55036c82c91bd56d8
https://techblogs.42gears.com/list-of-all-widely-used-adb-commands/
https://source.android.com/docs/core/tests/debug/understanding-logging
https://source.android.com/docs/core/architecture/configuration/add-system-properties
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/38efda47-3b75-483f-b9a2-66d9ada3719d/RUIDe8d33dbc10d94f318ee375f8f692b609.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/392bbbe3-8598-41df-a29f-fa6a4c4b7340/RUIDbe768115a64c4ca684c138e3e5bd3ad2.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/5a56f79f-379c-43fd-ae26-d6a911e9be16/RUIDafc40404a5cb49a79014cc8fadb207ce.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/39f41be4-dac1-4137-9bfe-16781a161381/RUID7c721c8d64cb473b8742a3c8b5eb8165.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/cebcaaac-d754-4c43-8842-b2e96e6d43dd/RUID1d9596375bda4ae8bfad8dfee88b8806.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/aff7bcb6-1225-4d69-ad3c-6f17f5f43765/RUID36b1c9caeedb43a2af3e490807e96201.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/059801f0-c2b7-4171-826f-41cadc8fa246/RUID27f634c5005e444ab9136d562d543243.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/aa282037-675b-4c0d-860e-3e0d43f1acfd/RUID334747d9d66842f6838cb6f73fd6934e.jpg

https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/c739d4e5-830b-4243-afc0-d4c805ae03b9/RUID2879f6b671b942be8b502abb5879cbd7.jpg
https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/145767158/d8ca8728-7a51-4a6b-89e8-814c3592838e/RUIDfde8a9d903314c4297b948bccf66ed0c.jpg

Enhanced Beta Prototype for Device Lexicon Extraction

This refined package builds on the skeleton you shared with production-ready improvements grounded in OCR and Android forensics best practices.

Key Architectural Improvements

Image Preprocessing Pipeline

The original skeleton defers OCR engine implementation. The enhanced version includes a full `preprocess.py` module that implements proven techniques for maximizing OCR accuracy on recovery logs. Recovery logs are often photographed at angles or under poor lighting, so preprocessing is critical. The pipeline supports grayscale conversion, binarization with configurable thresholds, denoising via morphological operations, CLAHE (Contrast Limited Adaptive Histogram Equalization) for contrast enhancement, and automatic deskew detection. Each step is optional and configurable, allowing you to profile your specific recovery images and toggle preprocessing based on accuracy impact.[1] [2]

Dual-Engine OCR with Fallback Strategy

Rather than a single NotImplementedError stub, the enhanced `ocr.py` supports both Tesseract and EasyOCR with automatic fallback. Tesseract is optimized for structured text (recovery logs), while EasyOCR provides a fallback if Tesseract fails or performs poorly. The OCRResult dataclass captures not just text, but confidence scores, raw line-by-line output, and engine metadata. This is essential for auditing which images had lower confidence and may need preprocessing tuning or manual review.[3] [4] [1]

Structured Android Property Parsing

The original `parser.py` uses simple regex matching on prefixes. The enhanced `AndroidLogParser` class understands the full Android property hierarchy, including read-only properties (ro.*), boot-time properties (ro.boot.*), vendor extensions (ro.vendor.*), and system properties (sys.*, net.*). It extracts timestamps where present and validates prefixes before creating PropEntry objects. This prevents false positives from unrelated text in the recovery output and groups properties by semantic category for downstream analysis.[5]

Intelligent Tokenization and Lexicon Analytics

The lexicon builder now performs meaningful token extraction with stopword filtering, minimum length thresholds, and noise detection heuristics. Instead of blindly splitting on whitespace, it differentiates between semantic tokens and garbage OCR artifacts (runs of numbers, special characters, etc.). The lexicon output includes token frequency distributions and per-prefix

property breakdowns, enabling you to identify device signatures and firmware markers programmatically.[1] [3]

Production CLI and Logging

The enhanced CLI uses argparse for clean argument handling, configurable logging to both console and file, and detailed error handling with graceful fallbacks. You can toggle preprocessing, select OCR engines, and save intermediate preprocessed images for inspection. This is crucial when debugging why certain recovery photos have lower OCR confidence.

Why These Additions Matter for Your Use Case

Recovery logs are text-heavy but visually degraded (phone camera angle, screen glare, variable lighting). Preprocessing with binarization and contrast enhancement can improve Tesseract confidence by 15–30%. The dual-engine strategy and confidence tracking let you audit which images need manual preprocessing or re-photography. Structured property parsing prevents the lexicon from getting polluted with garbage tokens, and tokenization analytics help you build device signatures (e.g., "all Samsung Galaxy devices have property X with pattern Y").[2] [5] [1]

Project Structure and Quick Start

The file above contains the complete enhanced prototype ready to drop into a GitHub repo. The pyproject.toml follows modern Python packaging with optional dev dependencies for testing. To get running: (1) initialize the folder structure, (2) install with `pip install -e ".[dev]"`, (3) place your recovery JPGs in `images/`, (4) run `device-lexicon --engine tesseract --preprocess`, (5) inspect `output/lexicons.json`.

Next Integration Points

Once this beta is working on your recovery images, consider:

- **Custom OCR Training**: If recovery logs use vendor-specific fonts or properties, train a Tesseract model on a small annotated sample for higher accuracy.[6]
- **Property Signature Mining**: Aggregate lexicons across multiple devices to identify firmware versions, manufacturer patterns, and security posture indicators.
- **Confidence-Based Filtering**: Filter properties with confidence < 0.7 and flag them for manual review or re-photograph.
- **Forensic Artifact Extraction**: Parse partition tables, boot state, security policy, and user profile data from the properties to reconstruct device state at recovery time.[7]

This prototype is testable, modular, and ready for iteration. The logging ensures you can diagnose OCR failures quickly, and the confidence tracking gives you visibility into data quality as you scale.

⁂

the github

I don't have a direct link to a GitHub repository for this prototype—it's a newly designed beta structure that I've provided as a complete, ready-to-use template rather than pointing to an existing repo.

Here's how to get started:

Option 1: Create Your Own Repo (Recommended)

1. Initialize a new repository:

```
mkdir device_lexicon_proto
cd device_lexicon_proto
git init
```

2. Create the folder structure and files from the template I provided (artifact)
3. Push to GitHub:

```
git remote add origin https://github.com/YOUR_USERNAME/device_lexicon_proto.git
git add .
git commit -m "Initial beta prototype"
git push -u origin main
```

Option 2: Reference Existing Similar Projects

If you're looking for reference implementations of OCR + property parsing pipelines, these GitHub projects have similar architecture patterns:

- **Tesseract Python wrappers**: Search GitHub for "pytesseract recovery logs" or "android property parser"
- **Android forensics tools**: Projects like `adb` wrappers, recovery log analyzers, and device property dumpers
- **OCR preprocessing pipelines**: Look for repos combining OpenCV + pytesseract for document scanning

Option 3: I Can Help You Initialize It

If you'd like, I can help you:

- Set up a bare GitHub repo and provide setup instructions
- Create a proper `.github/workflows` CI/CD pipeline (tests on push)
- Generate example recovery log samples for testing
- Build a test suite using pytest

Would you like me to help with any of these, or do you have a specific GitHub username/organization where you'd like to host this?

1. https://www.nutrient.io/blog/tesseract-python-guide/

2. https://stackoverflow.com/questions/9480013/image-processing-to-improve-tesseract-ocr-accuracy
3. https://nanonets.com/blog/ocr-with-tesseract/
4. https://blog.roboflow.com/ocr-data-extraction/
5. https://www.scribd.com/document/402869988/last-log
6. https://www.projectpro.io/article/how-to-train-tesseract-ocr-python/561
7. https://thebinaryhick.blog/2021/08/19/wipeout-detecting-android-factory-resets/
8. https://github.com/markuskuehnle/credit-ocr-system/blob/master/notebooks/2-ocr-based-text-extraction/README.md
9. https://www.youtube.com/watch?v=3BtLA75zKL0
10. https://unstract.com/blog/guide-to-optical-character-recognition-with-tesseract-ocr/