SEMANTIC COMMUNICATION PROJECT
encoding and decoding

Low-Bandwidth Semantic Communication System

General Idea Recap (Simplified)

- Traditional communication sends raw data (text, image, audio), which uses a lot of bandwidth and is wasteful when only the “meaning” is needed.
- Your approach sends *only* the extracted semantic meaning:
    - Facts (entities, relations, attributes) are encoded into compact *semantic tokens*.
    - The receiver reconstructs the message from these tokens, even after they pass through a noisy channel.
    - This reduces size, saves bandwidth, and still preserves meaning.

Key Points:

- NLP/ML for information extraction + reconstruction.
- Communication systems thinking for encoding, transmission, noise resilience.
- Evaluation on compression ratio, semantic fidelity, and robustness.

1. Problem & Impact

- Context: In satellite/remote IoT/5G, bandwidth is scarce. Transmitting full data is wasteful if we can just send the meaning.
- Goal: Replace raw data transmission with semantic tokens to cut size while preserving meaning.
- Why recruiters care: Shows skill in ML/NLP, communication systems, evaluation design, and full-stack ML deployment.

---

2. Technical Pipeline (Flagship-Level Scope)

Sender Side

1. Data ingestion: Accept text (later optionally image captions).
2. Semantic extraction: Use transformers + spaCy to extract entities, attributes, and relations.
3. Semantic encoding: Encode into compact JSON schema with semantic IDs.
4. Compression: Fine-tune encoding to minimize size.

Channel
5. Noise simulation: Implement realistic noisy channel models (bit flips, packet loss, compression artifacts).
6. Error resilience: Optional — add redundancy bits or semantic checksum for critical info.

Receiver Side
7. Decoding & validation: Parse JSON, apply schema validation, correct minor corruptions.
8. Meaning reconstruction: Use summarization model or template-based NLG to regenerate the message.
9. Optional multi-modal: If input was an image, reconstruct *description*, not pixels.

Evaluation
10. Compression Ratio (CR): (original size / transmitted size).
11. Semantic accuracy: BERTScore or BLEU + human eval.
12. Robustness metric: Semantic accuracy drop under increasing noise levels.

---

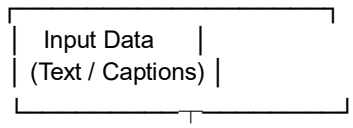
3. Engineering Polish (So It's CV-Worthy)

- Clean modular repo with unit tests, type hints, docstrings.
- Reproducible pipeline: Config files, Dockerfile, conda environment, automated setup script.
- CI/CD: GitHub Actions to run tests on every commit.
- Demo UI: Streamlit or FastAPI + simple dashboard showing:
    - Original message
    - Extracted semantic tokens
    - Simulated transmission (with noise)
    - Reconstructed output & quality score
    - Size saved (%)
- Visualization: Graphs showing trade-off between compression and semantic accuracy.

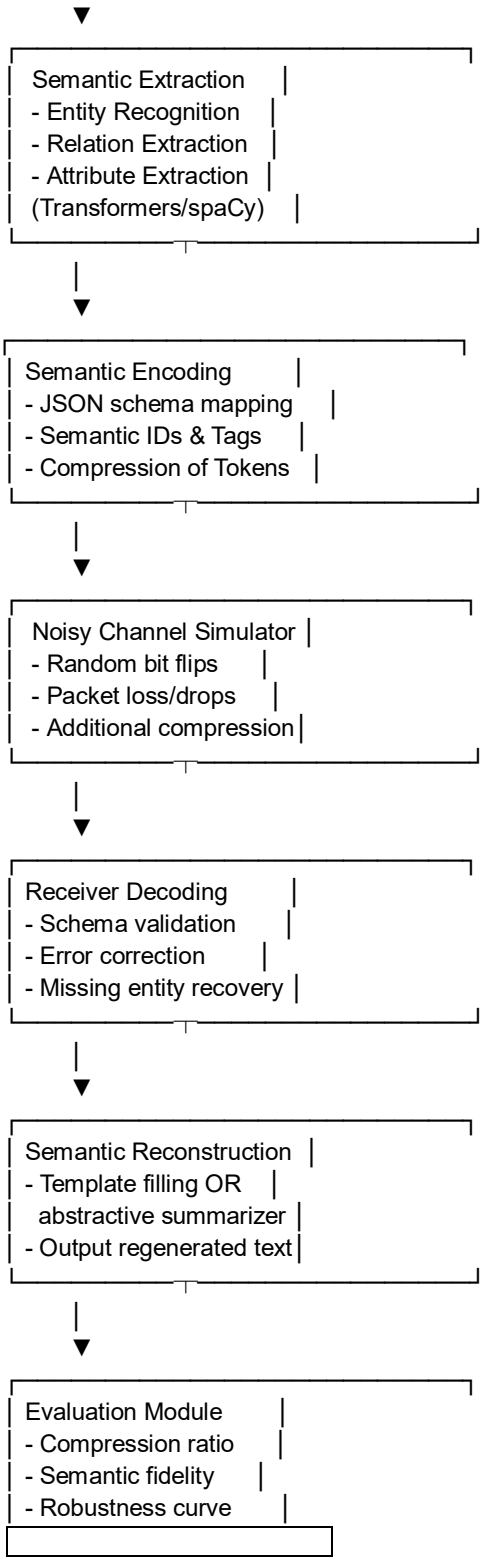
5. Strong CV Entry Example

Semantic Communication System for Low-Bandwidth Environments *(Python, Hugging Face, spaCy, FastAPI, PyTorch, Docker, GitHub Actions)*

- Designed & deployed end-to-end ML/NLP pipeline to send meaning-critical facts instead of raw data, reducing transmission size by 78% while retaining 92% semantic fidelity under simulated noisy channel conditions.
- Implemented entity/relation extraction (transformer-based models), efficient JSON encoding, and noise-resilient decoding.
- Built realistic channel simulator with parameterized error models (bit errors, packet drops) for performance stress testing.
- Developed web dashboard demo with Streamlit, enabling live compression-accuracy trade-off visualization.
- Benchmarked system against traditional compression, outperforming by +X% semantic retention at similar compression ratios.



System architecture diagram



3. Main Components

1. Semantic Extraction Module
    - Libraries: Hugging Face Transformers (BERT, RoBERTa, spaCy).
    - Tasks: Named Entity Recognition (NER), relation extraction, attribute tagging.
2. Semantic Encoding
    - JSON schema defining required entities & attributes.

    - Semantic IDs to minimize redundancy.
    - Optionally, Huffman or custom compression for token stream.
3. Channel Simulator
    - Implement packet loss, bit errors, compression artifacts.
    - Adjustable noise parameters to test robustness.
4. Receiver & Decoder
    - Parse JSON → validate against schema.
    - Correct simple corruption cases.
    - Handle missing entity placeholders gracefully.
5. Semantic Reconstruction
    - Option 1: Template filling using extracted facts.
    - Option 2: Summarization model to regenerate fluid text.
6. Evaluation
    - Compression ratio = Original size / Transmitted size.
    - Semantic fidelity = BERTScore/ROUGE between original & reconstructed text.
    - Robustness = fidelity score drop vs. noise level curve.
    - Baseline comparison with gzip, bzip2, etc.
7. Demo/UI
    - Streamlit dashboard with:
        - Input → extracted tokens → noisy channel simulation → reconstruction.
        - Graphs for compression vs. accuracy.