



Developing a Local Secret Redaction Prototype

From Model to Prototype: Integrating a Fine-Tuned Transformer Model

Bachelor Thesis

Spring Semester 2026

Author

Etienne Kaiser
etienne.kaiser@ost.ch

Author

Nico Heiniger
nico.heiniger@ost.ch

Advisor

Prof. Dr. Daniel Patrick Politze

Co-Advisor

Prof. Dr. Mitra Purandare

Co-Examiner

Ramon Schildknecht

Eastern Switzerland University of Applied Sciences (OST)
Department of Computer Science
Rapperswil, Switzerland

Revision Date: 12.06.2026

Abstract

Secret leakage in software artifacts remains a practical security problem, especially as users increasingly share content with external AI assistants and chat services. Existing local scanners are effective for well-known credential formats, but they struggle with unknown token formats, passwords and secrets that require contextual interpretation. This thesis studies a local hybrid approach to secret redaction, where the main objective is not fine-grained secret classification but the removal of sensitive values before text is shared outside the user’s trusted environment. The central hypothesis is that a multi-stage pipeline can improve redaction performance by combining the complementary strengths of rule-based detection, heuristic candidate generation and learned classification. The pipeline runs directly on the user’s device and does not require a GPU.

The developed pipeline combines deterministic, heuristic and learned components. Regular expressions detect known structured formats, entropy identifies random-looking candidates, and a fine-tuned GLiNER2 model detects context-dependent spans using a descriptive Secret label. Candidate spans generated from those stages are then merged, de-duplicated and passed to a LightGBM classifier that makes the final redaction decision. The classifier uses candidate-level features derived from the previous stages, including scores, entropy, shape and context. This separates recall-oriented candidate generation from precision-oriented final filtering. Access to a suitable real-world secret dataset hasn’t been granted within the project timeline, hence the experiments fallback to a synthetically generated dataset of software artifacts with annotated, non-functional secrets. The dataset is designed to represent the intended redaction workflow and includes logs, configuration files and source-code fragments. It also contains negatives such as hashes, dummy values and secret-like strings. The absence of final evaluation on real-world secrets is a central limitation of the thesis, because the reported results show controlled synthetic performance rather than externally validated real-world generalization.

The evaluation uses exact and partial matching with F2 being treated as the primary aggregate metric as it favors recall over precision, with importance towards secret leakage reduction. On the test set, the final hybrid (regex, entropy and model) pipeline achieves a F2 score of 0.9488 for exact matching (precision 0.9498 and recall 0.9486) and an F2 score of 0.9745 for partial matching. The F2 performance comparison supports the hypothesis that the redaction task benefits from combining multiple complementary signals. Rule-based detection and entropy provide useful candidate coverage, while the fine-tuned GLiNER2 model adds context-sensitive detection. However, using these candidate generators without a final decision layer leads to too many false positives. The strongest performance is achieved by the complete hybrid pipeline, where the LightGBM classifier filters the merged candidates and improves the balance between recall and precision. Analysis of classification errors indicates that remaining false positives are largely high-entropy negatives, such as hashes, while false negatives are primarily associated with low-entropy passwords and URLs containing embedded credentials.

Keywords: redaction, secrets, detection, GLiNER2, NER, LightGBM, Tauri, Shannon entropy, regex

Acknowledgements

We would like to thank Prof. Dr. Daniel Patrick Politze, Prof. Dr. Mitra Purandare and Ramon Schildknecht for their guidance and patient feedback throughout the project. We are also grateful to the INS Institute for Network and Security at OST for providing the initial setting and the discussions that shaped the direction of this work. Finally, we thank our families and friends for their support during the more intense weeks of the thesis.

Table of Contents

1	Introduction	1
1.1	Initial Situation	1
1.2	Problem Statement	2
1.3	Objective	3
1.4	Scope and Limitations	3
1.5	Thesis Structure	4
2	Related Work	5
2.1	Secret Leakage in Source Code Repositories	5
2.2	Rule-Based and Heuristic Detection	5
2.3	Machine Learning and Deep Learning Approaches	6
2.4	Dataset Quality and Evaluation Challenges	6
2.5	Research Gap	6
3	Background and Foundations	8
3.1	Secret Types and Software Artifacts	8
3.2	Regular Expressions	9
3.3	Shannon Entropy	10
3.4	Named Entity Recognition	11
3.5	GLiNER and Flexible Span Detection	11
3.6	Fine-Tuning and Transfer Learning	12
3.7	Evaluation Metrics	12
4	Methodology	13
4.1	Evaluation Strategy	13
4.2	Notion of Improvement	14
4.3	Value-Level vs. Label-Level Evaluation	14
4.4	Exact and Partial Matching	14
4.5	Baselines	14
4.6	Experimental Procedure	15
5	Architecture and Prototype	16
5.1	C4 Architecture Overview	16
5.1.1	C1: System Context	16
5.1.2	C2: Container View	16
5.1.3	C3: Backend Redaction Components	18
5.2	Hybrid Redaction Pipeline	20
5.2.1	Detection Stages	20
5.2.2	Candidate Merging and Final Decision	21
5.2.3	Masking and Output Generation	21
5.3	Design Iterations of the Pipeline	21
5.3.1	V1: Regex and Entropy Baseline	22
5.3.2	V2: GLiNER2-based Span Detection	23
5.3.3	V3: Hybrid Candidate Generation	23
5.3.4	V4: Candidate Classification and Final Scoring	24
5.3.5	Final Pipeline Selection	25
5.4	Regex Catalog Overview	26
6	Dataset and Experimental Setup	27
6.1	Dataset Requirements	27
6.2	SecretBench and Access Delay	27
6.3	Synthetic Dataset Generation	28
6.4	Dataset Format	29
6.5	Exploratory Data Analysis	30
6.5.1	Dataset Composition	30

6.5.2	Secret-Type Coverage	30
6.5.3	Artifact and Secret Characteristics	31
6.6	Negative Examples	32
6.7	De-duplication and Leakage Prevention	33
6.8	Train, Validation and Test Set	33
6.9	Dataset Limitations	34
6.10	Experimental Setup	34
6.11	Baseline Implementations	34
6.12	Model Comparison Setup	35
6.13	GLiNER2 LoRA Fine-Tuning Setup	36
6.13.1	Experiment Subsets	37
6.13.2	Experiment Results	38
6.13.3	Five-Fold Robustness Check	39
6.13.4	Interpretation	40
6.13.5	Limitations of the Training Experiments	40
6.14	Meta-Classifer Training	41
6.15	Hybrid Ablation Setup	42
6.16	Threats to Reproducibility	43
7	Results	44
7.1	Baseline Performance	44
7.2	Model Comparison	45
7.3	GLiNER2 Fine-Tuning Results	48
7.3.1	Training Behavior	48
7.3.2	Threshold Sensitivity	49
7.3.3	Final Synthetic Test Results	50
7.3.4	Best Configuration	51
7.4	Candidate Classifier Results	51
7.5	Meta-Classifer Results	51
7.5.1	Classifier Performance	51
7.5.2	Threshold Selection	54
7.6	Pipeline Results and Ablation	56
7.7	Stage-Wise Contribution Analysis	57
7.8	Error Analysis	58
7.8.1	Thresholded Partial Matching	58
7.8.2	Per-Category Performance	59
7.8.3	Error Pattern Analysis	61
7.8.4	Qualitative Error Examples	62
7.8.5	Effect of Entropy Threshold	63
7.9	Runtime Observations	63
8	Discussion	65
8.1	Interpretation of the Results	65
8.2	Comparison with the Predecessor Prototype	65
8.3	Comparison with Related Work	66
8.4	Engineering Trade-offs	66
8.5	Research Trade-offs	67
8.6	Threats to Validity	67
8.7	Possible Use Case	68
9	Conclusion	70
9.1	Answer to the Research Question	70
9.2	Future Work	70
9.3	Closing Remarks	71

Bibliography	72
List of Abbreviations	75
Glossary	76
List of Figures	78
List of Tables	79
A AI Tool Usage Disclosure	80
B Tooling	81

1 Introduction

In modern software development, the volume of textual artifacts that can carry sensitive information has grown massively. Source files, configuration files, container manifests, log files, notebooks and even some commit messages routinely contain credentials such as API keys, authentication tokens, database connection strings and passwords. When such artifacts are shared with a colleague over chat, pasted into an online assistant or pushed to a public repository, the embedded secrets can leak. Once a secret is exposed, an attacker can use it to impersonate the legitimate owner, exfiltrate further data or move laterally inside the target system. Empirical studies of public repositories report millions of leaked secrets across many programming languages and file types [1], [2]. The problem is not new, Sinha et al. demonstrated already in 2015 that hardcoded API keys appear in a substantial fraction of public projects and proposed early countermeasures based on keyword search and program slicing [3].

The mainstream response to this problem are secret scanners. Tools such as TruffleHog [4], Gitleaks [5], GitGuardian [6] and the secret-scanning service of GitHub [7] combine large catalogs of provider-specific regular expressions with Shannon-entropy heuristics that flag random-looking strings as suspicious [1], [2]. These approaches are fast, deterministic and easy to deploy, but they suffer from two well-documented weaknesses. First, regular expressions match by surface form only, so they cannot tell a real key apart from a placeholder, a UUID used as a database identifier or a hash inside a comment. The result is a high rate of false positives that undermine the trust of developers and forces manual intervention. Second, regular expressions miss everything that does not match a known pattern, in particular self-chosen passwords and tokens whose format has not yet been cataloged. Recent large-scale studies report precision and recall figures that vary widely across detectors and across secret categories, and identify low-entropy chosen passwords as the most challenging case [1], [2].

Machine-learning-based approaches have begun to close this gap. Rahman et al. fine-tuned LLaMA-3.1 and Mistral models on top of a regex pre-filter and reported an F1 of 0.985 in binary secret classification on the SecretBench dataset [1]. Transformer-based encoders such as BERT and GraphCodeBERT have been used in similar pipelines to embed the context surrounding a candidate string [2]. These results show that contextual reasoning can substantially reduce the false positive rate of pure pattern matching. They also expose a deployment problem: large language models with several billion parameters cannot be run locally on a developer laptop, and sending source code to a cloud-based detection service is precisely the kind of disclosure that the system is supposed to prevent.

Compact encoder-based architectures offer a way out of this dilemma. GLiNER [8] is a bidirectional transformer with fewer than 500M parameters that performs open-type named entity recognition by matching entity-type prompts against text spans in a shared latent space. Its successor GLiNER2 [9] extends this idea to a multi-task, schema-driven framework. Both variants are designed to run on CPU, which makes them candidates for fully offline secret detection.

1.1 Initial Situation

The work presented in this thesis is the direct continuation of an earlier project by us, the study work “Offline Multimodal AI Redaction of Sensitive Data in Audio and Software Artifacts” [10]. In that project, an offline web application was built that combines Faster-Whisper [11] for multilingual speech-to-text transcription. Microsoft Presidio’s [12] framework handled a trivial NER-based spaCy model for sensitive data detection in combination with a custom regex engine leading to masking. The architecture is summarized in Figure 1 and ran inside Docker containers on the user device. While the prototype demonstrated that high-quality offline PII detection is possible (recall close to 100% for most PII categories), the secret-detection part exhibited a clear performance gap: per-category recall ranged from 41.7% on plain API keys

to 100% on UUIDs and IPv6 addresses and a non-trivial fraction of secrets were only partially redacted, leaving the remaining characters available for reconstruction by an attacker, such as crypto seed phrases.

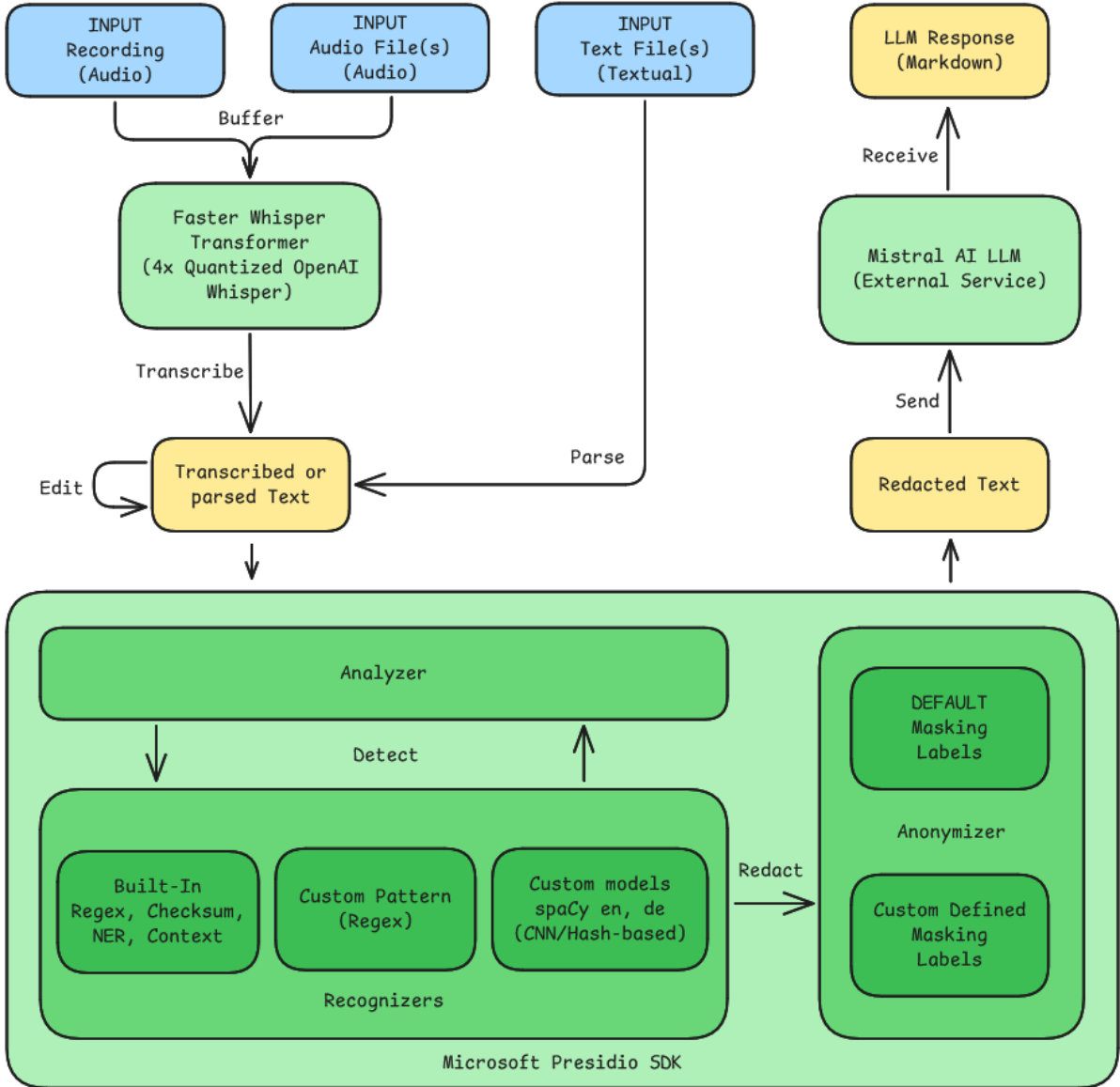


Figure 1: Architecture of the predecessor prototype.

The two main lessons from that work motivate the present thesis directly. First, regex-only secret detection is inherently limited, it cannot reason about context and over-redacts non-secret tokens that happen to look random. Second, classical NER models such as spaCy are not trained on cybersecurity-specific entity types and provide no measurable improvement on secrets. Both conclusions point in the same direction, namely that a model dedicated to secrets and trained on a representative dataset is needed in addition to, not instead of, the rule-based stage.

1.2 Problem Statement

Secrets in textual artifacts share several properties that make them harder to detect than classical PII. They are short, often randomly generated, and they lack the syntactic regularity of names, addresses or phone numbers that classical NER models exploit. Formats change over time and new providers are added regularly, which means that any static regex catalog is by construction incomplete. The most challenging

category, self-chosen passwords, follows no syntactic pattern at all and can only be recognized from the surrounding context (for example the literal “password=” next to it).

A reliable detection system therefore has to satisfy three requirements simultaneously:

- Run entirely offline on a user device, so that no input data ever leaves the local environment. This is a non-negotiable consequence of the use case: a tool that uploads files to a cloud service in order to redact them is self-defeating.
- Neither fast, deterministic pattern matching nor context-aware model inference is sufficient on its own, the former misses anything outside its fixed catalog, the latter is too slow and imprecise to be the sole detection mechanism.
- Produce structured output (typed placeholders, span offsets, source of the detection), since false redactions break the surrounding text in unpredictable ways and false negatives leak the very information the tool was supposed to protect.

1.3 Objective

The objective of this thesis is to design, train and evaluate a hybrid, fully local secret redaction pipeline that combines a fine-tuned GLiNER transformer model with a regex-and-entropy detection, and to integrate it into an installable Tauri desktop application. The pipeline produces a redacted version of the input text.

The work pursues the following concrete goals:

- **Model selection:** Compare several pretrained encoder NER baselines (different GLiNER variants, GLiNER-P11, GLiNER2 and some models from other model-families) on fitting dataset and pick the strongest baseline as the starting point for fine-tuning.
- **Fine-tuning:** Fine-tune the chosen model with a controlled exploration of learning rate, batch size, optimizer, and document the effect of each on the F2 score.
- **Hybridization:** Quantify how much regex pre-filtering and Shannon-entropy filtering add on top of the fine-tuned model alone, as a precision-oriented post-filter.
- **Deployment:** Package the resulting model inside a Tauri application that loads the model on first start, performs redaction locally and writes redacted output. Optionally a shortcut for the redaction would be interesting to demonstrate a possible real-world usecase.

The thesis is explicitly *not* aimed at producing a production-grade scanner. It is a research and engineering prototype, meaning the focus is on quantifying the contribution of each stage and on demonstrating feasibility, not on covering every possible secret format.

Research Question

Does combining rule-based methods (regex, entropy) with a neural NER model improve detection performance compared to using either approach alone?

1.4 Scope and Limitations

The thesis focuses on secrets in textual software artifacts. The exact secret categories evaluated are documented in [Section 3.1](#). PII redaction and speech-to-text processing are outside the main scope of this thesis. They were addressed in the preceding semester thesis and are not continued here because the bachelor thesis investigates the distinct problem of detecting access-related secrets in software artifacts. A generic PII detector may redact names or phone numbers, but it does not by itself answer whether application secrets can be reliably identified and removed.

Due to extended access delays and queue times for the SecretBench benchmark [13], a synthetically generated dataset was utilized for model training, selection, and final evaluation within the required project timeframe. The final prototype is evaluated on English-language synthetic test data, while multilingual and real-world evaluations are deferred to future work.

Several limitations apply to the conclusions that can be drawn. First, the Tauri application is treated as a proof of concept for local usage, not as a production-ready security product. Second, claims regarding detection quality are limited to the evaluated datasets and configurations. In particular, results obtained on synthetic data cannot by themselves establish performance on real repositories or production logs. Third, manual labeling of false positives in the UI is out of scope, the system cannot learn from user corrections at runtime. Finally, the model is run in full precision on CPU, quantization, distillation and on-device inference optimization are interesting but not covered here.

1.5 Thesis Structure

The remainder of the thesis is organized as follows: [Section 2](#) discusses the state of the art in secret detection, including rule-based, machine-learning-based and hybrid approaches, and identifies the research gap that this work addresses. [Section 3](#) introduces the formal foundations the reader needs in order to follow the rest of the document, in particular regular expressions, Shannon entropy, transformer-based NER and the GLiNER architecture. [Section 4](#) develops the methodology, presents the influence diagram of the experimental design, formulates the hypotheses and explains the evaluation metrics. [Section 5](#) describes the architecture of the Tauri application and the role of the model inside it. [Section 6](#) covers the generated dataset, the intended but unavailable SecretBench benchmark, the exploratory data analysis, the preprocessing pipeline and the experimental setup for the baselines, the model-based stages and the hybrid pipeline. [Section 7](#) reports the experimental results and answers the research question. [Section 8](#) places the results in context and discusses the deployment trade-offs of the prototype. [Section 9](#) concludes the thesis and outlines future work. The appendix contains the AI-usage disclosure, the project management documentation, additional dataset analysis, the regex catalog and installation details.

2 Related Work

This chapter studies the literature on secret detection along four dimensions: empirical studies of secret leakage, rule-based and heuristic approaches, machine-learning-based detection, and dataset quality. The closing section identifies the research gap that the present thesis addresses.

2.1 Secret Leakage in Source Code Repositories

The exposure of secrets such as API keys, authentication tokens, private keys, and credentials is a long-known security issue. Public version-control platforms have amplified the problem because once a secret is committed, even a quick force-push usually does not undo the leak, the value is already cached by integrated services, automation bots and adversaries. Sinha et al. [3] provide a systematic analysis. Using keyword search, regular expressions and slicing, they showed that hardcoded secrets occur frequently in public Java projects and proposed mitigations integrated into the developer workflow, such as pre-commit hooks. Their conclusion that pure pattern matching cannot handle the diversity of secret formats has been confirmed by every subsequent study known by us.

Recent work has scaled this analysis to platforms beyond GitHub. Zhou et al. [2] scanned more than 80M files across GitHub, PyPI and WeChat and reported that secret leakage is no longer only affecting source files but also occurs in configuration files, package manifests, log files and documentation. Their cross-platform benchmark consisting of 11826 labeled secrets confirms that platform bias is a real issue. Patterns and conventions differ between ecosystems, and detectors trained on GitHub alone underperform on other sources. Rahman et al. [1] report similar observations on a large GitHub sample with 49 programming languages and 311 file types, again with log and configuration files as a substantial fraction of all leaks.

2.2 Rule-Based and Heuristic Detection

Industrial scanners largely rely on rule-based approaches. TruffleHog, Gitleaks and GitGuardian use catalogs of several hundred to several thousand specific regular expressions [14], complemented by Shannon-entropy heuristics for tokens that do not match any known pattern. The strength of this approach is its determinism: a given input always yields the same output, which makes the scanners suitable for CI/CD integration and for security audits where reproducibility matters.

The weakness of pure pattern matching is also well documented. Rahman et al. [1] report that regex-based detection can produce a high rate of false positives, and Zhou et al. [2] observe that recall drops sharply on unstructured or human-chosen secrets. KEYSSENTINEL [2] attempts to mitigate these problems by combining a catalog of around 910 regular expressions with prefix matching, semantic filtering and a machine-learning-based classifier for human-chosen secrets. The authors report improved precision and recall over both TruffleHog and Gitleaks on their cross-platform benchmark, but the overall pipeline remains limited to a manual selected regex list, with the well-known maintenance cost that this implies. Comparative evaluations of secret-detection tools further show that reported findings differ substantially between tools, which complicates a direct comparison of detector quality [15].

Entropy is the second pillar of rule-based detection. Random-looking strings tend to have high Shannon entropy because their characters are roughly uniformly distributed, while natural text has a much lower entropy. Tools such as Yelp’s detect-secrets [16] ship a HighEntropyString plugin and Gitleaks [5] combines entropy with regex rules. As we discuss in Section 3.3, entropy is computationally cheap and effective on cryptographic secrets but fundamentally blind to context. Thus, a long random hex string inside a Git commit hash and an AWS secret key have similar entropy, yet only one of them is a secret.

2.3 Machine Learning and Deep Learning Approaches

Early machine-learning work on secret detection used hand-crafted features such as character distribution, length and entropy, and trained classical classifiers such as random forests. These approaches improved over pure regex but remained trivial: they captured only statistics of the candidate string and ignored the surrounding text. The shift to deep learning was driven by the realization that the most reliable signal for many secrets, in particular passwords, is contextual, the literal password= next to the candidate is more informative than the candidate itself.

Two lines of work followed. The first applies large language models to the task. Rahman et al. [1] combine a regex pre-filter (768 patterns) with a fine-tuned LLaMA-3.1 8B classifier and report an F1 of 0.985 in binary secret classification on SecretBench, as well as competitive results with Mistral-7B. The authors explicitly investigate smaller models for local deployment, motivated by the same privacy concern that drives the present thesis. However, models of 7 - 8B parameters remain impractical for the target deployment context: they exceed the memory budget of standard user hardware, autoregressive inference is too slow for interactive redaction of multi-page documents on CPU-only devices, and bundling multi-billion parameter weights into a desktop installer is not feasible. The second approach adapts encoder-based transformers such as BERT and GraphCodeBERT [2] to embed the candidate together with a window of surrounding code, and shows that contextual embeddings reduce the false-positive rate significantly compared to feature-engineered baselines.

A separate line of research focuses on compact, open-type NER models. GLiNER [8] introduces a bidirectional transformer architecture that performs span prediction in parallel and matches entity-type prompts in a shared latent space with the spans. The result is a model that handles arbitrary entity types with fewer than 500M parameters and runs efficiently on CPU. GLiNER2 [9] generalizes this idea to a schema-driven, multi-task framework that supports NER, classification and hierarchical extraction with a single 205M parameter model. These properties make GLiNER and GLiNER2 strong candidates for a fully local secret-detection pipeline running on CPU only user devices.

2.4 Dataset Quality and Evaluation Challenges

Majority of reviewed published results on secret detection rely on SecretBench [13], a benchmark of roughly 97000 candidate secrets and 15000 manually verified true secrets collected from 818 public GitHub repositories. SecretBench has been fundamental in moving the field, but it has a known weakness: the dataset contains a large number of duplicate and near-duplicate samples. Soltaniani and Ghafari [17] demonstrated that this duplication leads to data leakage between training and test sets and inflates evaluation metrics. After de-duplication they observed that the Matthews Correlation Coefficient of a random forest dropped from 0.89 to 0.65, an LSTM from 0.92 to 0.77, and GraphCodeBERT by roughly seven percentage. These numbers indicate that previously reported results overestimate generalization and that any new study should de-duplicate before evaluating.

2.5 Research Gap

The literature shows three converging trends. First, pure rule-based detection is reaching a ceiling and the next gains come from contextual, learned models. Second, the strongest ML-based results so far rely on large language models that cannot be run on a developer laptop and that, in their typical deployment, would require sending the input to a remote service. Third, dataset quality matters, results that are not de-duplicated cannot be taken at face value.

This thesis addresses the gap that follows from these observations. It takes a compact, encoder-based model in the GLiNER family, fine-tunes it on the generated synthetic dataset, and integrates it into a fully local Tauri prototype. To the best of our knowledge, no prior work combines (i) a fine-tuned GLiNER

variant for secrets, (ii) regex and entropy as complementary detectors with quantified contributions, and (iii) an installable desktop application that runs the resulting model locally and completely offline. [Table 1](#) summarizes how the present work positions itself relative to existing tools and studies.

System / Study	Regex	Entropy	ML model	Local	Open-source
TruffleHog [4]	Yes	Yes	No	Yes	Yes
Gitleaks [5]	Yes	Yes	No	Yes	Yes
GitHub Secret Scanning	Yes	Partial	No	No	Partial
GitGuardian	Yes	Yes	Light	No	No
KEYSENTINEL [2]	Yes	Implicit	Yes (filter)	Yes	Partial
Rahman et al. [1]	Yes	No	LLaMA / Mistral	Cloud	Code yes, model heavy
Kaiser & Heiniger 2025 [10]	Yes	No	spaCy NER (PII)	Yes	Planned
This thesis	Yes	Yes	Fine-tuned GLiNER2	Yes	Planned

Table 1: Comparison of secret-detection tools and detection strategies.

3 Background and Foundations

This chapter introduces the concepts required to understand the proposed pipeline and its evaluation. It focuses on secret types, deterministic and statistical detection signals, neural span detection, fine-tuning, and the metrics used in the experiments.

3.1 Secret Types and Software Artifacts

In this thesis, the term *secret* refers to sensitive authentication or access-related values that should not be exposed outside a controlled environment. Examples include API keys, access tokens, passwords, private keys, database connection strings, webhook URLs, and cloud provider credentials. Unlike general personally identifiable information (PII), secrets are not primarily sensitive because they identify a person, but because they may grant access to systems, services, data or infrastructure.

In this work, software artifacts include source code, configuration files, environment files, log files, shell scripts, deployment files, and text snippets copied from development workflows. These artifacts are especially relevant for redaction because developers often share them with external systems such as issue trackers, support channels, documentation tools, or AI assistants. If secrets are included in such inputs, they may unintentionally leave the local machine or the controlled organizational environment.

The structure of secrets differs from classical natural-language entities. Some secrets follow defined patterns, such as credentials with known prefixes or encoded private-key blocks. Others are less structured and may only appear suspicious because of their entropy, length, surrounding context, or variable name. For example, a value assigned to `DATABASE_URL`, `Authorization`, or `api_key` may be sensitive even if the value itself does not follow a known format. A further challenge is that non-secret identifiers, hashes, UUIDs, or test values can resemble secrets. Secret detection is therefore a mixed problem: structured cases can often be detected deterministically with rules, while ambiguous cases require additional heuristic or contextual interpretation.

For this reason, the thesis distinguishes between categories of secrets and secret-like values. [Table 2](#) provides representative examples of the secret types considered relevant for software artifacts and highlights why no single detection approach is sufficient for all categories.

Secret type	Example occurrence	Detection challenge
API key or access token	Source code, environment files, HTTP headers	Known prefixes exist for some providers, while custom formats may be unknown.
Password	Configuration files, connection code, deployment scripts	The value may only be recognizable through its surrounding key or context.
Database connection string	.env files, application configuration, logs	One value can embed multiple sensitive components, such as a username and password.
Private-key material	Key files, configuration snippets, CI/CD variables	The secret can span multiple lines and typically has a structured envelope.
Webhook URL	Integration configuration and deployment scripts	It resembles an ordinary URL but may authorise actions on an external service.
High-entropy token	Logs, generated configuration, copied debugging output	It may not have a known prefix and can be confused with non-secret hashes or identifiers.
Non-secret lookalike	Identifiers, UUIDs, checksums, placeholder values	It resembles a secret but does not grant access and should ideally remain visible.

Table 2: Examples of secret types and secret-like values.

The primary objective of the developed pipeline is redaction rather than fine-grained classification. If a sensitive value is correctly removed under a generic SECRET label, the practical security objective is achieved even when a more specific category such as API_KEY or PASSWORD is not assigned. Consequently, the evaluation focuses primarily on whether the original secret value remains visible after processing, treating partial redaction as a secondary concern, a case where the sensitive value is only partly removed. This decision is described in detail in [Section 4.3](#).

3.2 Regular Expressions

A regular expression (regex) is a formal pattern over an alphabet that defines a set of strings called a regular language. In the classical formulation due to Kleene, the operators of concatenation, alternation and Kleene-star are sufficient to describe every regular language [18]. Modern programming language libraries extend this core with conveniences such as character classes, lookahead and back-references.

For secret detection, regex are attractive for three reasons. They are deterministic, which means that the same input always yields the same output, an essential property for compliance audits and CI/CD pipelines. They are fast, most engines compile a pattern in linear time. Finally, they are simple to share. A pattern is a single string, easy to store in a configuration file and easy to review (but complex variations are possible).

The limitations are equally well known. A regular expression matches by surface form only. It cannot express semantic constraints such as “this 32 character hexadecimal string is a secret only when it appears as the value of a key called api_key”. Pattern formats evolve over time, which means that a regex catalog is by construction incomplete and needs continuous maintenance. Empirical studies report large false-positive rates when regex is used as the only detector [1], [2].

Table 3 shows three representative patterns used in industrial scanners. The first captures AWS access keys, which start with the well-known AKIA prefix and have a fixed length. The second matches GitHub personal access tokens of the ghp_ family. The third shows a generic high-entropy string pattern that requires an entropy filter to be useful in practice, the regex alone matches far too many strings.

Secret type	Regular expression (simplified)	Comment
AWS Access Key ID	AKIA[0-9A-Z]{16}	Fixed prefix, 20 characters total. Very precise.
GitHub PAT (classic)	ghp_[0-9A-Za-z]{36}	Prefix introduced in 2021. Very precise.
Generic high-entropy	[A-Za-z0-9+/=]{32,}	Matches base64 but needs entropy filter to be useful.

Table 3: Examples of regular expressions for common secret formats.

In this thesis, regular expressions plays the role as a high recall candidate generator that flags everything that looks like a known pattern secret. The detailed pattern catalog used in the prototype is described in Section 5.4.

3.3 Shannon Entropy

Shannon entropy measures the average information content of a discrete random variable [19], [20]. For a string s of length n over an alphabet α , the empirical entropy is

$$H(s) = - \sum_{c \in \alpha} p_c \log_2 p_c$$

where p_c is the relative frequency of character c in s . The result is in bits per character: a uniformly random string over an alphabet of $|\alpha|$ symbols approaches $\log_2(|\alpha|)$ bits per character, while a string with strong character regularities has substantially lower entropy.

Two intuitions matter for secret detection. First, cryptographic secrets are by design close to uniform random, so they tend to have high empirical entropy. AWS secret keys, private SSH keys and base64-encoded tokens routinely show entropies above four bits per character. Second, natural language text has much lower entropy because letters are not uniformly distributed and follow strong contextual rules. English text is around 1.0 - 1.5 bits per character once context is taken into account [20], even character-level entropy without context is typically below 4.5 bits.

Entropy as a detection signal is computationally cheap, language agnostic and complementary to regex. It is also blind to context. A long Git commit hash, a UUID, a hash of a CI artifact and a real AWS secret key have similar entropies, and an entropy filter cannot tell them apart without further information. This is the reason why every entropy-based scanner in practice combines entropy with at least a length threshold, often a context check, and sometimes a classifier. Figure 2 shows the entropy distribution of secrets and non-secrets in the synthetic dataset. Overlap in the middle range motivates additional methods on top of the entropy filter.

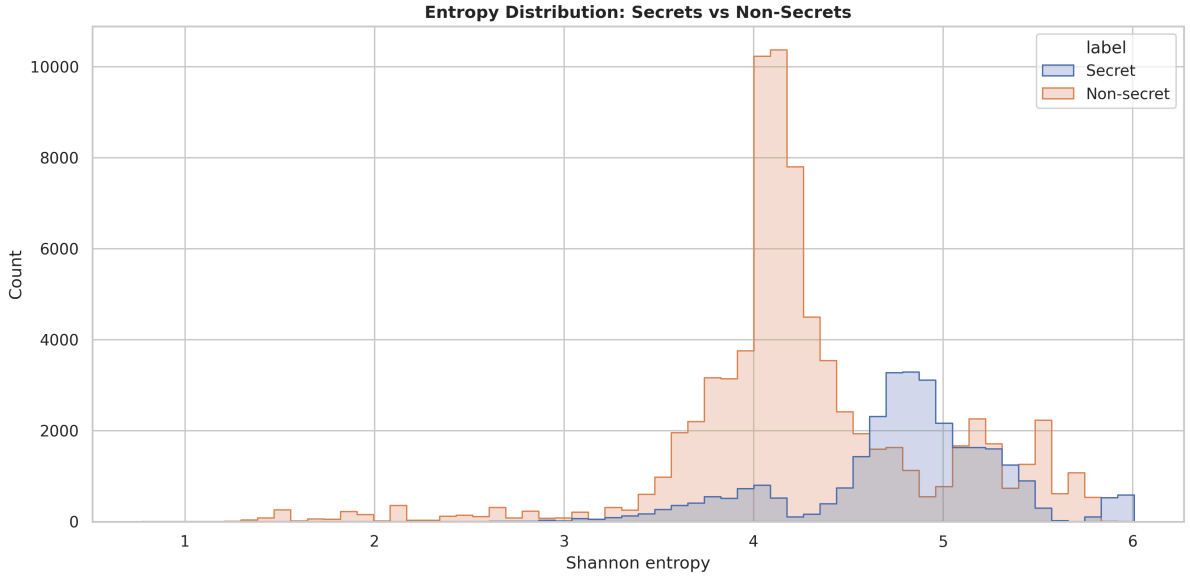


Figure 2: Shannon entropy distribution of secrets and non-secret strings in the dataset.

3.4 Named Entity Recognition

Named Entity Recognition is the task of locating spans in text and assigning them to a finite set of types. Classical NER datasets such as CoNLL-2003 [21] distinguish between persons, locations, organizations and other entities. The task has been studied for more than two decades and modern systems have moved from hand-crafted features and conditional random fields to neural sequence taggers and most recently, transformer-based span classifiers [22], [23].

Two design choices are relevant in modern NER architectures. First, *open-type* NERmodels accept the entity type as a runtime argument rather than as a fixed output dimension. This allows the same model to detect new entity types without retraining, which is useful in domains where the set of relevant types evolves over time. Second, *span-based* prediction directly enumerates candidate spans and assigns them a type score, instead of predicting a tag at every token. Span-based architectures are evaluated at the value level rather than the token level, and they natively handle nested entities where one entity is embedded inside a longer surrounding span.

Transformer-based encoder models are the current standard for NER. A bidirectional encoder processes the input in parallel, producing a contextual representation for every token that takes the entire input window into account [22]. Models such as BERT, RoBERTa and DeBERTa, all in the encoder family, dominate the NER leaderboards. Their main limitation for secret detection is that they require a fixed entity-type set that has to be defined at training time.

3.5 GLiNER and Flexible Span Detection

GLiNER [8] addresses these limitations directly. It is an open-type NER model based on a bidirectional transformer encoder. At inference time the user supplies a list of entity types as natural language prompts, for example ["api key", "password", "private key"]. The model encodes the prompts and the input text in the same backbone, enumerates candidate spans up to a configurable maximum length, and computes a similarity score between every span and every prompt in a shared latent space. A span is predicted as an entity of a given type if the corresponding similarity exceeds the threshold.

The prompt-based interface allows experimentation with new entity types without retraining the full model: zero-shot evaluation on a new type often yields non-trivial recall, and a few hundred fine-tuning examples can be enough to bring the F1 into a usable range.

GLiNER2 [9] extends the architecture into a multi-task framework with around 200M parameters. The user provides a structured schema that describes both the extraction targets and the desired output format. The model can perform NER, classification and hierarchical extraction within a single forward pass. For secret detection this is useful because we can express both the binary “is this a secret?” decision and the multi-class “which kind of secret is it?” decision as a single schema. We discuss in [Section 7.2](#) the practical question of whether to use GLiNER, GLiNER2, or other related encoder NER models.

3.6 Fine-Tuning and Transfer Learning

Pretrained encoder models contain general-purpose representations of language that have to be adapted to a specific task before they can be used directly. Fine-tuning is the process of continuing the training of such a pretrained model on a task-specific dataset, typically with a much lower learning rate than the original pretraining and for a small number of epochs [24]. The intuition is that the lower layers of the encoder have already learned useful syntactic and semantic features and only need small adjustments, while the upper layers and the task head are more task-specific and benefit from more aggressive updates.

Several variations of fine-tuning are commonly used. Full fine-tuning updates every parameter of the model and is the most expressive option, but it is also the most expensive in memory and the most prone to overfitting on small datasets. Layer-wise unfreezing starts from a fully frozen model and gradually unfreezes the upper layers as training progresses, which is a popular middle ground [24]. Parameter-efficient methods such as LoRA [25] insert low-rank adapters into the attention layers and train only the adapters, keeping the original weights frozen. LoRA reduces the need of memory dramatically and is especially attractive for the GLiNER family on CPU.

In the present thesis we evaluate LoRA fine-tuning of the GLiNER variants with the choice driven by the available training and time resources. Concrete hyperparameter ranges are documented in [Section 6.13](#).

3.7 Evaluation Metrics

The standard metrics for span-level NER are precision, recall and F1, defined over true positives (TP), false positives (FP) and false negatives (FN):

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1 is the special case $\beta = 1$ of the more general F-measure, which lets the relative weight of recall be adjusted through a parameter β :

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

Setting $\beta = 2$ yields the F2-score, which weights recall four times as strongly as precision ($\beta^2 = 4$). For secret redaction the cost of a false negative (a secret that leaks because it was not redacted) is higher than the cost of a false positive (a non-secret that was redacted), so F2 matches the objective better than F1. This thesis therefore reports F2 as the primary aggregate metric and F1 as a secondary metric for comparability with the detection literature.

4 Methodology

This chapter describes how the thesis evaluates whether a hybrid secret redaction pipeline improves detection performance compared to rule-based or model-based approaches used in isolation. It defines the evaluation strategy, the meaning of “better” in the context of secret redaction, the matching rules and metrics used for evaluation, the baseline systems, the experimental procedure, and the main threats to validity.

4.1 Evaluation Strategy

The evaluation strategy is designed to compare several detection approaches under the same conditions. Each approach is evaluated on the same test data and with the same matching rules. This ensures that differences in precision, recall, and F2-score are caused by the detection strategy rather than by changes in the dataset or evaluation procedure.

The overall evaluation is guided by the influence diagram in Figure 3. The diagram separates the main factors that can affect the final detection quality into input and data factors, preprocessing decisions, model-related parameters, decision thresholds, and evaluation metrics. Positive and negative arrows express expected effects that are subsequently investigated through experiments.

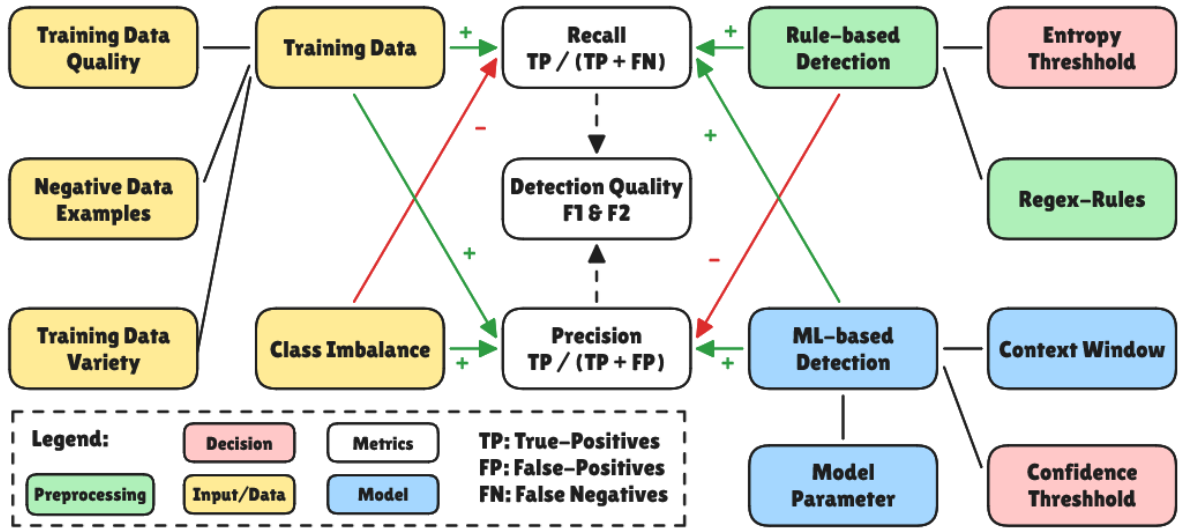


Figure 3: Influence diagram of the evaluation strategy.

The diagram highlights two main detection paths. On the rule-based path, the rule catalog and entropy threshold influence which candidates are proposed. On the ML-based path, the selected model, context representation, and confidence thresholds influence which spans are proposed and accepted. Training-data quality, variety, and negative examples can affect the learned components. Finally, precision and recall jointly determine F1/F2-score.

The purpose of the diagram is not to prove these relationships directly. Instead, it serves as a structured overview of the assumptions behind the experiments. For example, lowering a confidence threshold may increase recall but reduce precision, while adding more regex rules may improve coverage but also introduce false positives.

4.2 Notion of Improvement

In this thesis, one detection approach is *better* than another when it leaves fewer true secrets visible in the output without raising over-redaction to an impractical level. This follows directly from the redaction goal: an undetected secret (a false negative) stays exposed, whereas an unnecessary redaction (a false positive) only reduces the readability of the shared text.

The evaluation therefore prioritizes the metrics in the following order:

- **Recall** is the most security-critical metric, because every missed secret stays leaked.
- **F2-score** is the primary aggregate metric, because it weights recall more strongly than precision. (detailed in [Section 3.7](#))
- **Precision** remains relevant, because excessive redaction makes the output less useful.
- **F1-score** is reported as a secondary aggregate metric for comparability with the detection literature.

Concretely, a configuration with slightly lower precision but substantially higher recall is preferred, as long as the resulting over-redaction stays within a practically acceptable range.

4.3 Value-Level vs. Label-Level Evaluation

The methodology is designed around the practical goal of redaction rather than fine-grained secret classification. The primary evaluation question is therefore whether a sensitive value remains visible after processing, not whether the predicted label matches exactly. In value-level evaluation a detection is successful if the secret value is removed or masked, regardless of the label assigned. In label-level evaluation the predicted category must also match the expected category, for example `API_KEY`, `PASSWORD` or `DATABASE_URI`, which is relevant when the goal is classification rather than redaction. This thesis focuses on value-level evaluation, and the final pipeline uses a single generic `Secret` label for the main redaction decision. If a GitHub token is detected and redacted under that label, the redaction still prevents leakage even though the exact category is not preserved.

4.4 Exact and Partial Matching

The evaluation distinguishes between exact and partial matching. Exact matching is the stricter criterion and requires the full secret value to be detected and redacted. It is used as the primary success definition because a partially visible secret may still be sensitive or reconstructable.

However, partial matching is also reported as a secondary analysis. Partial matches occur when only part of a secret value is detected or redacted. This is relevant because not all partial detections have the same practical effect.

For this reason, partial matches are analyzed by measuring how much of the original secret value was covered by the predicted redaction span. This provides a more nuanced view than a purely binary exact-match evaluation. Nevertheless, partial matches are not treated as fully successful detections in the primary metric unless explicitly stated. If several predicted spans overlap with the same ground-truth secret, the union of the overlapping spans is used to compute the covered fraction. This avoids double-counting overlapping detections from multiple pipeline stages.

4.5 Baselines

The hybrid pipeline is evaluated against a set of baselines that represent the main detection strategies investigated in this thesis. Each baseline is a reference point for interpreting the final result, and every

baseline and hybrid configuration is evaluated on the same test data with the same matching rules (Table 4).

Configuration	Detection strategy	Reported evidence
Regex only	Structured known-format detection	Rule-catalog version and measured metrics
Entropy only	Pattern-independent high-entropy heuristic	Entropy threshold and measured metrics
Regex + entropy	Combined non-neural detection	Measured metrics
GLiNER2 only	Neural span detection	Label schema, threshold, fine-tuned adapter.
Secrets-Sentinel [26]	Candidate-level classification signal	Used as an optional feature, not as a standalone detector
Full hybrid pipeline	Proposed combined approach	Final components, metrics
Hybrid minus one stage	Stage-wise contribution analysis	Reported per ablated stage

Table 4: Baseline and hybrid configurations.

The rule-based baselines are regex only, entropy only and their union. The model-based reference points are GLiNER2 span detection and the candidate-level classifier. The zero-shot model comparison in Section 7.2 selects the neural component of the pipeline, it is model selection for the proposed pipeline rather than an exhaustive benchmark of all secret-detection models. The hybrid configurations combine the rule-based and model-based stages, and in the later pipeline generations the candidates produced by regex, entropy, GLiNER2 and the optional Secrets-Sentinel signal are turned into feature values and passed to a meta-classifier that makes the final redaction decision for each candidate.

4.6 Experimental Procedure

The experiments follow seven stages:

1. **Dataset preparation.** The dataset is generated, validated and converted into a common JSONL format containing the input text, secret values, labels and character positions.
2. **Exploratory data analysis.** The dataset is analyzed for artifact length, secret-type distribution, entropy distribution, file-type distribution, duplicate values and the number of secrets per artifact.
3. **Baseline evaluation.** Regex-only, entropy-only and combined regex-entropy configurations establish the performance of rule-based detection.
4. **Model comparison.** CPU-compatible candidate models are compared on detection quality, runtime, training effort and suitability for the local prototype.
5. **Fine-tuning.** The selected model is fine-tuned for secret detection, covering label design, context-window size, confidence thresholds and training-data composition.
6. **Hybrid pipeline construction.** The rule-based and model-based stages are combined, and candidate spans are merged, deduplicated, scored and passed to the final decision component.
7. **Final evaluation and error analysis.** The final pipeline is evaluated against the baselines, and false positives, false negatives, per-category behavior and runtime are analyzed to answer the research question.

F2 is the primary aggregate metric because false negatives leave secrets visible, while F1, precision and recall are also reported. The dataset split, experiment parameters, hardware environment and reproducibility measures are documented in Section 6.10.

5 Architecture and Prototype

The architecture is intentionally simple: a Tauri shell, a Python sidecar that exposes the redaction pipeline over a local HTTP socket, and a thin web front-end. The chapter documents each layer in turn, from the high-level system boundary down to the backend redaction engine, and also covers the regex catalog, the de-duplication strategy, and the redaction schema.

5.1 C4 Architecture Overview

The prototype architecture is documented using the C4 model [27]. The purpose of the C4 views is not to describe every implementation detail, but to make the system boundary and main internal components understandable at different levels. The architecture is presented in three levels: the system context view defines the prototype boundary and its interaction with the user, the container view decomposes the prototype into its main runtime parts, and the component view focuses on the backend redaction engine, where the hybrid detection and final candidate decision logic are implemented.

5.1.1 C1: System Context

The C1 diagram in Figure 4 shows the prototype as a single system from the user’s perspective. The primary actor is the user, who provides input text, source-code fragments, configuration files, logs, or prose, to the local redaction prototype, which processes it and returns redacted text.



Figure 4: System context diagram of the redaction prototype.

External chatbots, tools, and editors are intentionally not modeled as integrated systems. They belong to the wider usage context but are not directly connected to the prototype, the user remains responsible for transferring the redacted output externally. This distinction defines the trust boundary of the research artifact: the prototype acts as a local privacy-preserving layer before any external communication takes place.

5.1.2 C2: Container View

The C2 diagram in Figure 5 decomposes the prototype into its main runtime parts. The Tauri application [28] forms the desktop shell. Inside this boundary, a JavaScript-based frontend provides the user interface, while a Python backend exposes the redaction functionality through FastAPI [29] and coordinates the processing workflow.

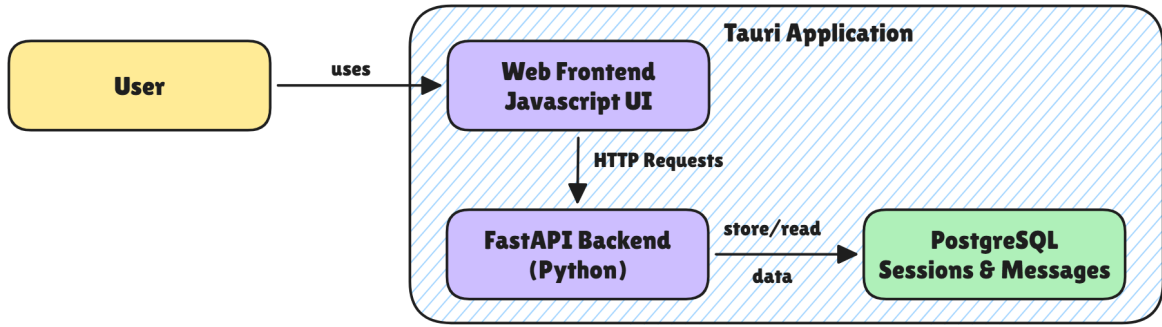


Figure 5: Container diagram of the redaction prototype.

Tauri packages the web front-end with a Rust core into a small native desktop application. Rather than bundling a full browser, it uses the operating system’s native webview, keeping the installer to a few megabytes (plus model weights). The Rust core handles file system access, process management, and inter-process communication, the front-end is rendered inside the webview and communicates with the core through Tauri’s command system. On startup, the Rust core spawns the Python sidecar, registers it in the app state, and tears it down on window close. The full build pipeline is illustrated in Figure 6.

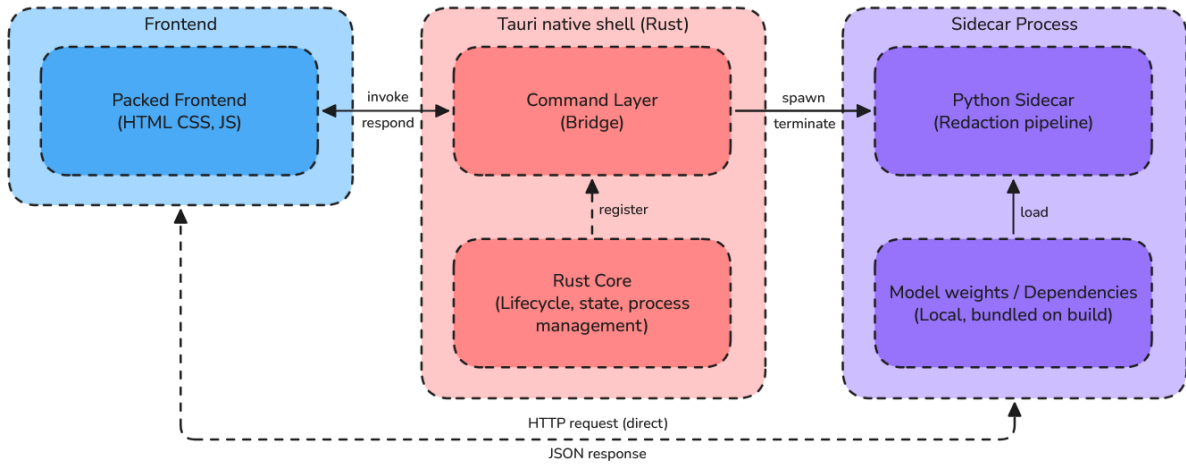


Figure 6: Tauri application architecture and build pipeline overview.

The frontend receives pasted or manually entered text, displays the redacted result, and provides workflow actions such as triggering redaction or copying the output. Detection logic is not implemented in the frontend, instead, redaction requests are forwarded over HTTP to the backend through a thin Tauri command layer.

The backend is a Python service that acts as the orchestration layer. It receives requests from the frontend, invokes the redaction pipeline, and returns the processed output. This separation keeps the user interface independent of the model and pipeline implementation and makes the redaction engine straightforward to test outside the desktop shell. Communication between frontend and backend follows a simple request-response pattern: the frontend sends input text, the backend processes it locally, and the frontend displays the returned result.

The Python sidecar is bundled with PyInstaller [30] into a single self-contained executable. The build script downloads the model weights, freezes the Python dependencies, copies the binary into the Tauri sidecar location, and produces the final installer. Current build targets are Windows (.msi and .exe), with macOS and Linux support planned.

A persistence component stores sessions and messages independently of the redaction logic, so the detection pipeline does not depend directly on the storage implementation. The left panel in Figure 7 shows the original .env artifact containing a credential-bearing database URL, a plain-text password, and

an AWS access key. The right panel shows the redacted output, where all three detected values have been replaced by [SECRET]. Non-sensitive lines such as APP_ENV and APP_PORT are preserved unchanged.

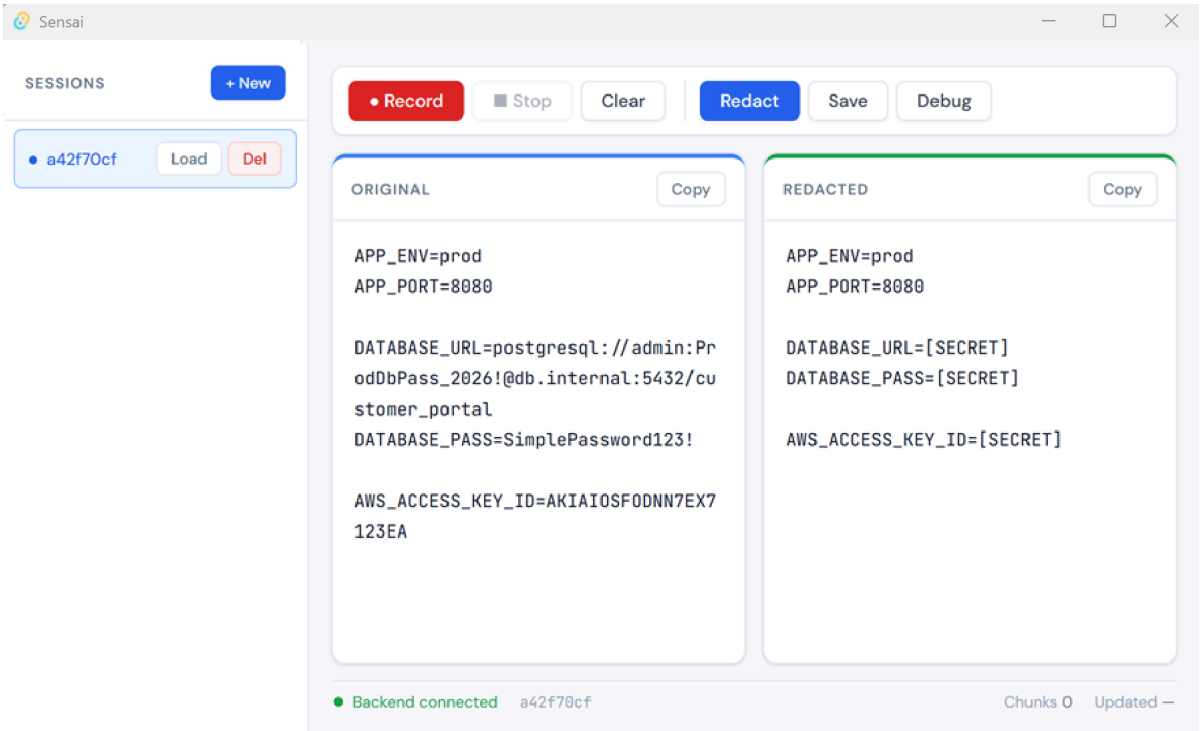


Figure 7: The Sensai prototype during a redaction session.

5.1.3 C3: Backend Redaction Components

The C3 component diagram in Figure 8 focuses on the internal structure of the backend and, in particular, on the redaction pipeline. This view is the most relevant one for the scientific contribution of the thesis, as it shows how candidate generation, candidate merging, final classification, and masking are kept separate.

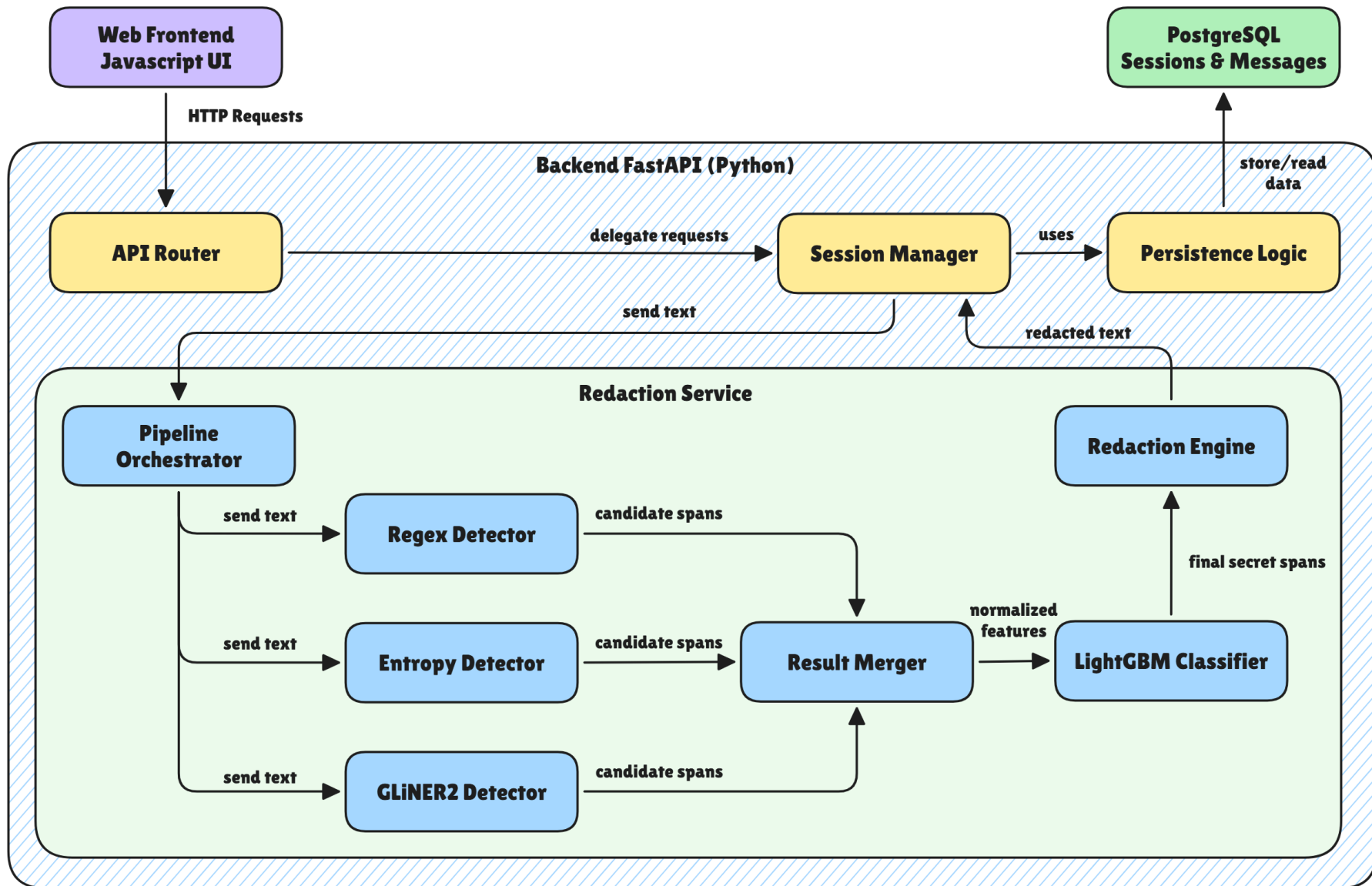


Figure 8: C4 component diagram of the backend redaction engine.

The API router is the backend entry point. It receives requests from the frontend, validates the request shape, and delegates work to the internal orchestration layer. The session manager then coordinates execution: it receives input from the API layer, invokes the pipeline orchestrator, triggers persistence if requested, and returns the final redacted text. It is responsible for the backend workflow, but not for the individual detection algorithms.

The pipeline orchestrator manages the redaction pipeline itself. It calls the individual candidate-generation stages, passes their outputs to the result-merging stage, enriches the resulting candidates with features, invokes the final decision layer, and forwards confirmed detections to the masking engine.

The redaction service combines the candidate outputs of the regex, entropy, and GLiNER2 stages and prepares them for the final decision layer. The result merger normalizes these detections, resolves overlaps, removes duplicates, collects features, and produces the unified candidate representation. The final binary classifier then decides which candidates should be retained as true secrets, separating broad, recall-oriented candidate generation from the more precision-oriented final decision. Confirmed detections are passed to the masking engine, which replaces the corresponding spans with a placeholder [SECRET] and returns the redacted text to the frontend.

5.2 Hybrid Redaction Pipeline

The hybrid redaction pipeline is the central technical component of the prototype. It processes text locally and produces a redacted output in which detected secrets are replaced.

The pipeline combines multiple different detection and binary classification mechanisms. Rule-based stages are intended to identify structured or high-entropy values efficiently and deterministically, while the model-based stage is intended to recognize context-dependent secrets that do not follow a known format and have a low entropy value. The pipeline is structured into separate stages so that the contribution of individual components can be evaluated independently in the later experiments.

5.2.1 Detection Stages

The pipeline uses four candidate signals because secrets in software artifacts differ substantially in structure and context. Some follow provider-specific formats, others are only recognizable from surrounding variable names, syntax or statistical properties.

Regex-based detection applies a catalog of manually selected patterns to the input text. It targets secrets with recognizable structure, such as tokens with known prefixes, private-key envelopes and connection-string formats. It is deterministic and computationally inexpensive, but it only detects formats represented by an existing rule, so unknown or weakly structured secrets remain undetected. The catalog is summarized in [Section 5.4](#).

Entropy-based detection flags candidate values with high character diversity that resemble randomly generated credentials or tokens, complementing regex by targeting values that do not match a known format. It is useful for unknown token formats but produces false positives, because hashes, identifiers, encoded values and UUID-like strings can show similar statistical characteristics without being secrets.

GLiNER2 span detection is the neural component. Unlike regex and entropy, it uses surrounding text to identify secrets whose sensitivity depends on context, for example a value next to a key such as `password`, `api_key` or `authorization` that does not match any known pattern. The model uses the generic SECRET label rather than fine-grained categories, which reflects the redaction goal: a value is protected once it is removed, even without a specific category.

Candidate classification (Secrets-Sentinel) is an optional additional signal. Unlike the detection stages, it does not search the input for new spans, it estimates whether an already extracted candidate is likely to be a real secret. Because the detection stages aim for broad coverage, they also surface hashes, identifiers,

UUIDs and placeholder values. For each candidate, Secrets-Sentinel receives the candidate value and its surrounding context, so a value next to terms such as `password`, `token` or `api_key` can gain stronger support than an equally random-looking value in a non-sensitive context. Its output is not the final decision, it is combined with the other stages in the final decision step and is extracted as a feature for the meta-classifier (Section 7.5).

5.2.2 Candidate Merging and Final Decision

The individual detection stages may identify identical or overlapping spans. For example, a token with a known prefix may be detected both by a regex rule and by the neural model, while an embedded credential may overlap with a larger connection-string candidate. The pipeline therefore requires an explicit merging and conflict-resolution step before masking is applied.

In the initial pipeline design, overlapping candidates were merged and duplicate detections were removed before final redaction. This reduces repeated processing of identical values and prevents multiple placeholders from being produced for the same secret.

In the extended hybrid pipeline, candidate-level information collected from the individual stages is used to support the final decision. Such information may include whether a candidate was detected by regex, whether it exceeded the entropy threshold, whether GLiNER2 predicted a matching span, whether Secrets-Sentinel provided supporting evidence, and the corresponding confidence or score values.

A LightGBM classifier [31] was evaluated as a meta-classifier for combining these signals into a final decision on whether a candidate should be redacted. This approach allows the pipeline to learn which combinations of signals are most indicative of true secrets instead of relying only on manually defined thresholds.

5.2.3 Masking and Output Generation

After the final candidate decision, all confirmed secret spans are passed to the masking stage. The masking stage replaces detected sensitive values with a generic placeholder `[SECRET]`, while preserving the surrounding non-sensitive text. This produces an output that can be reviewed by the user and subsequently copied or shared without exposing the detected secret values.

The use of a generic placeholder is consistent with the primary goal of the prototype. The objective is to prevent sensitive values from leaving the local environment rather than to provide a complete semantic classification of every credential type. More specific labels may still be retained internally for analysis or for deterministic regex detections, but they are not required for successful redaction in the primary evaluation.

The redacted output is returned to the user-facing application for review and further use. The local desktop application therefore demonstrates the intended workflow of the thesis: potentially sensitive software artifacts are processed and redacted locally before being manually transmitted to an external service.

5.3 Design Iterations of the Pipeline

The hybrid pipeline was developed iteratively. Only iterations that changed the conceptual detection strategy are described here, minor bug fixes, individual regex additions, and threshold adjustments are treated as implementation refinements rather than separate pipeline generations. The four iterations are summarized in Table 5, and each subsection ends with a short learning summary that motivates the next step toward the final selection in Section 5.3.5.

Iteration	Configuration	Purpose	Limitation or reason for next step
V1	Regex and entropy	Establish a deterministic non-neural baseline.	Limited contextual understanding, coverage depends on patterns and thresholds.
V2	GLiNER2-based span detection	Investigate context-aware secret detection with a generic label.	Must be compared empirically with structured-rule detection, may introduce uncertain candidates.
V3	Regex, entropy, and GLiNER2 candidate generation	Combine complementary candidate sources.	Requires overlap resolution and a principled final acceptance decision.
V4	Hybrid candidates plus additional classifier signal and meta-classifier	Use candidate features to decide final redactions.	Selected as the final configuration after threshold tuning on validation data.

Table 5: Conceptual design iterations of the secret-redaction pipeline.

5.3.1 V1: *Regex and Entropy Baseline*

The first generation used regular expressions and entropy as rule-based candidate generators. Regex rules targeted known secret formats, while the entropy stage proposed random-looking strings that did not necessarily match a predefined pattern. The resulting candidates were de-duplicated and then passed to masking (Figure 9).

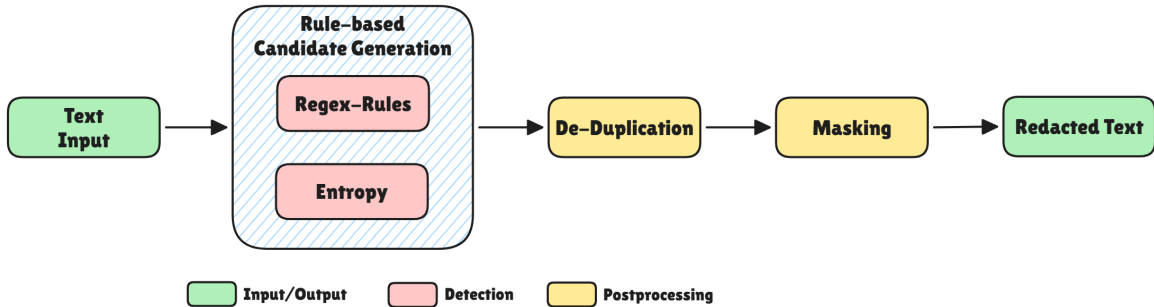


Figure 9: Rule-based redaction pipeline using regex and entropy

This generation established the initial baseline. The advantage of the design was that it was simple, deterministic and explainable. Regex rules are especially useful for provider tokens, private-key material and credential-bearing URLs. Entropy adds a broader heuristic signal for random-looking strings that do not match a specific pattern.

However, the rule-based generation also showed clear limitations. Regex only detects patterns that are explicitly defined in the catalog, while entropy introduces many false positives when used too broadly. On the synthetic test set, the best exact-matching rule-based baseline was regex-only, with precision 0.6132, recall 0.6180 and F1 0.6156. Adding entropy increased coverage in some settings, but it also added noise.

Learning summary. Rule-based detection is useful but not sufficient on its own: regex misses everything outside its catalog, and entropy trades recall for many false positives. The pipeline needs a context-sensitive component for secrets that follow no known pattern, which motivates V2.

5.3.2 V2: GLiNER2-based Span Detection

The second generation used a single NER-based detection stage. Text input was processed by the GLiNER2 model, and detected spans were passed directly to the masking step (Figure 10).

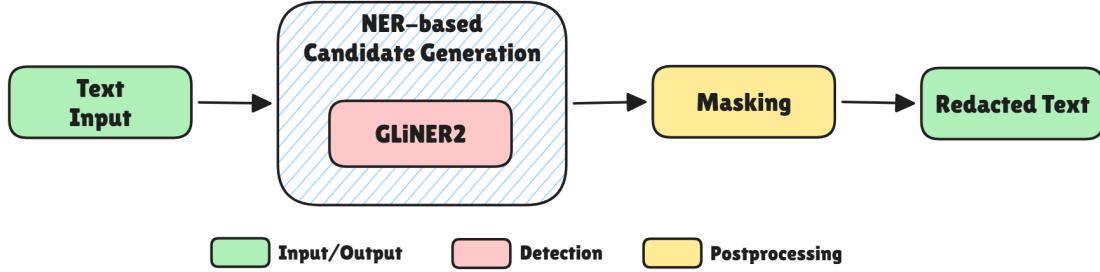


Figure 10: NER-based redaction pipeline using GLiNER2.

This generation tested whether a fine-tuned span-detection model could solve the redaction task more generally than fixed rules. The advantage of this design was that GLiNER2 can use context and is not limited to a predefined regex catalog. This is particularly relevant for passwords, unknown token formats and values whose sensitivity depends on nearby variable names or configuration keys.

The model-based design improved the ability to detect context-dependent spans, but it still had limitations. The fine-tuned GLiNER2 model achieved high precision (exact-matching precision 0.9978) but recall of only 0.8080 on the synthetic test set (exact-matching F1 0.8929), which is not sufficient for a security-oriented redaction workflow.

Learning summary. GLiNER2 is a strong, high-precision detector, but on its own it is too conservative for redaction, and it covers different failure modes than the rules. V3 therefore combines the rule-based and model-based signals instead of choosing one.

5.3.3 V3: Hybrid Candidate Generation

The third generation combined rule-based and NER-based candidate generation. The input was processed by the regex and entropy stages in parallel with the GLiNER2 stage. Their outputs were then merged and deduplicated before masking (Figure 11).

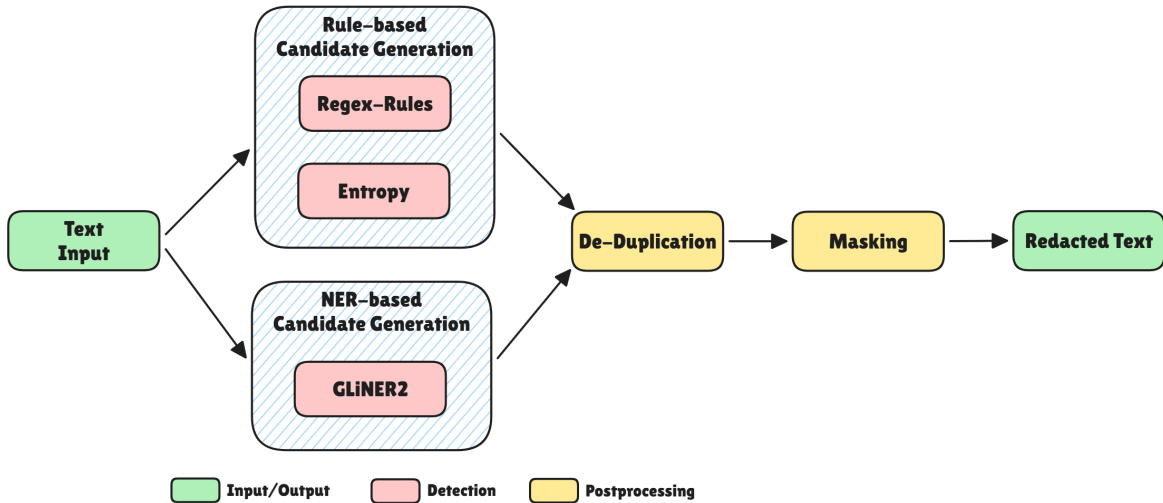


Figure 11: Hybrid candidate generation using rule-based and NER-based detection.

This generation was introduced because the previous two approaches had complementary strengths. Regex provided deterministic support for known formats, entropy added broad candidate coverage for random-looking values, and GLiNER2 contributed context-sensitive span detection. Combining them increased the chance that a true secret was proposed by at least one stage.

The experiment confirmed this expectation, but also revealed the main weakness of a simple candidate union. The no-classifier hybrid run achieved very high exact-matching recall on the synthetic test set (0.9649), meaning that most target secrets were covered by at least one candidate stage. However, precision collapsed to 0.2125, resulting in exact-matching F1 0.3483. The pipeline produced too many candidates to be useful as a final redaction system.

Learning summary. Combining detectors maximizes recall (0.9649) but produces far too many false positives (precision 0.2125). Broad candidate generation alone is not enough, the architecture needs a separate decision layer that filters candidates without sacrificing recall. This leads to V4.

5.3.4 V4: Candidate Classification and Final Scoring

The fourth generation introduced a learned candidate-level decision layer. Regex, entropy and GLiNER2 still generate candidates, but these candidates are now merged, deduplicated and enriched with features before a LightGBM meta-classifier decides whether each candidate should be retained. Secrets-Sentinel was evaluated as an optional binary-classification signal, but the final preferred variant does not require it (Figure 12).

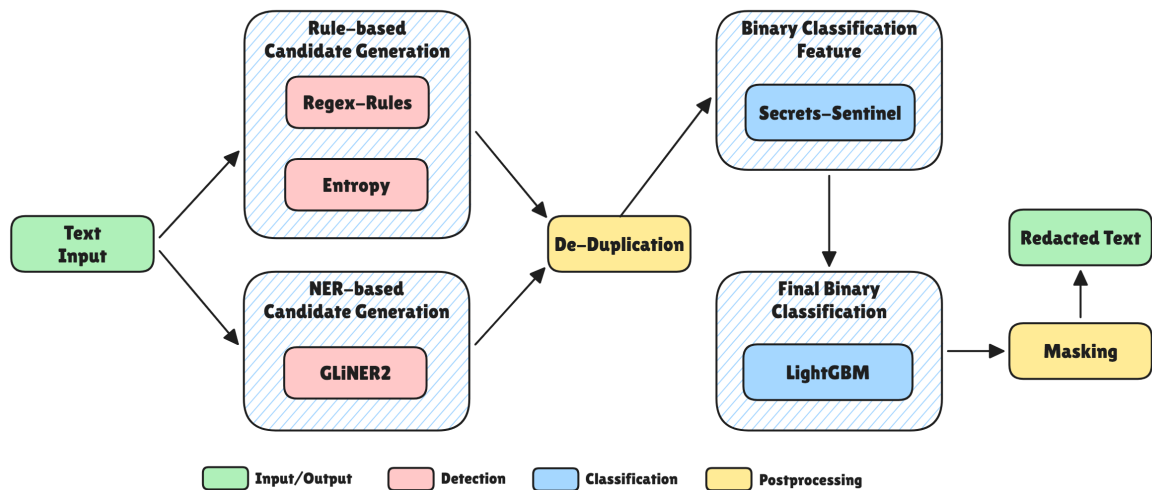


Figure 12: Final hybrid pipeline with LightGBM meta-classifier and masking.

This generation addresses the central problem of Generation 3 (Figure 11). Instead of treating every generated candidate as a final detection, the pipeline separates recall-oriented candidate generation from precision-oriented candidate filtering. The meta-classifier receives interpretable features such as detector provenance, candidate entropy, candidate length, shape features, GLiNER2 confidence, regex support and context indicators.

The final experiments showed that this separation is the central architectural improvement. On the synthetic test set, the selected classifier-free meta-classifier pipeline reached exact-matching precision 0.9498, recall 0.9486 and F1 0.9492, and a partial-matching F1 of 0.9748. Compared with V3, the meta-classifier retained most of the recall while reducing false positives from 21,626 to a few hundred.

Learning summary. Separating broad candidate generation from a learned keep/drop decision is the central improvement: it preserves V3's recall while restoring precision (exact-matching F1 from 0.3483 to 0.9492). This is the configuration selected as the final pipeline.

5.3.5 Final Pipeline Selection

Figure 13 provides an overview of the runtime processing flow of the pipeline. It receives a text input, such as a source-code fragment, configuration file, log excerpt, or just a simple message. Several detection stages then analyze the input and produce possible candidate spans, that may contain a secret.

The rule-based stages search for known secret patterns and suspicious high-entropy values. In parallel, GLiNER2 analyzes the input for context-dependent secret spans using the configured secret label schema. The resulting candidates are subsequently merged and deduplicated before the candidates are provided to Secrets-Sentinel, which does a binary classification per candidate. A final decision is then made with the trained LightGBM [31] classifier for each candidate. Confirmed detections are passed to the masking stage, which produces the redacted output.

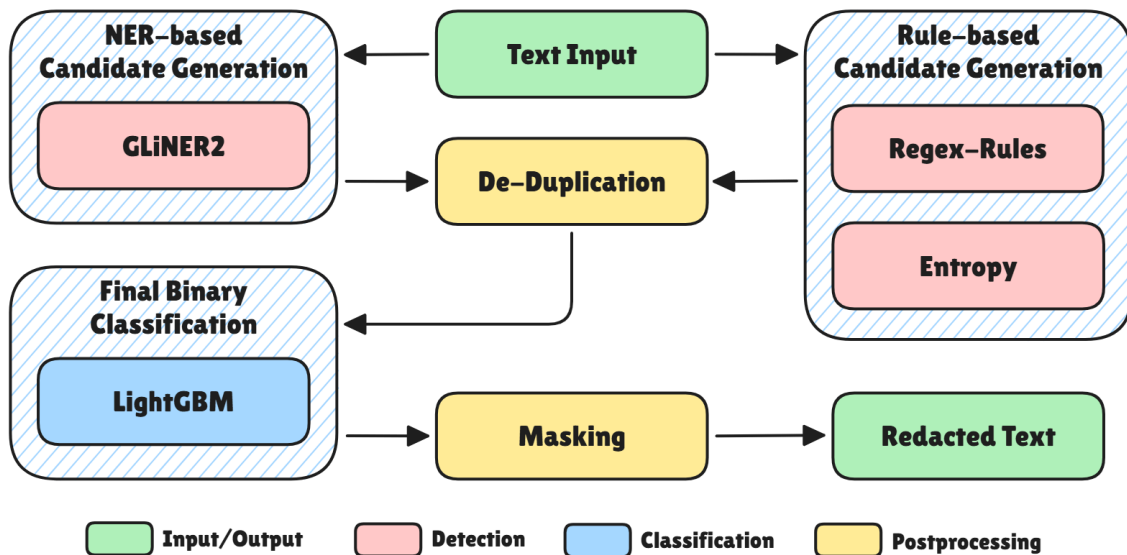


Figure 13: Runtime process flow of the hybrid redaction pipeline.

5.4 Regex Catalog Overview

The regex-based stage uses a conservative catalog of 36 patterns as shown in [Figure 14](#). It does not attempt to match every random-looking value or every keyword-value assignment, because overly broad patterns would increase false positives and would make the synthetic benchmark easier to solve through generator-specific rules. The catalog therefore serves two purposes. First, it provides a deterministic and lightweight baseline for known secret formats. Second, it contributes structured candidate signals to the later hybrid pipeline. The labels associated with the rules are retained for analysis and catalog organization, but the primary redaction evaluation still treats all confirmed sensitive values under the generic SECRET objective described in [Section 4.3](#).

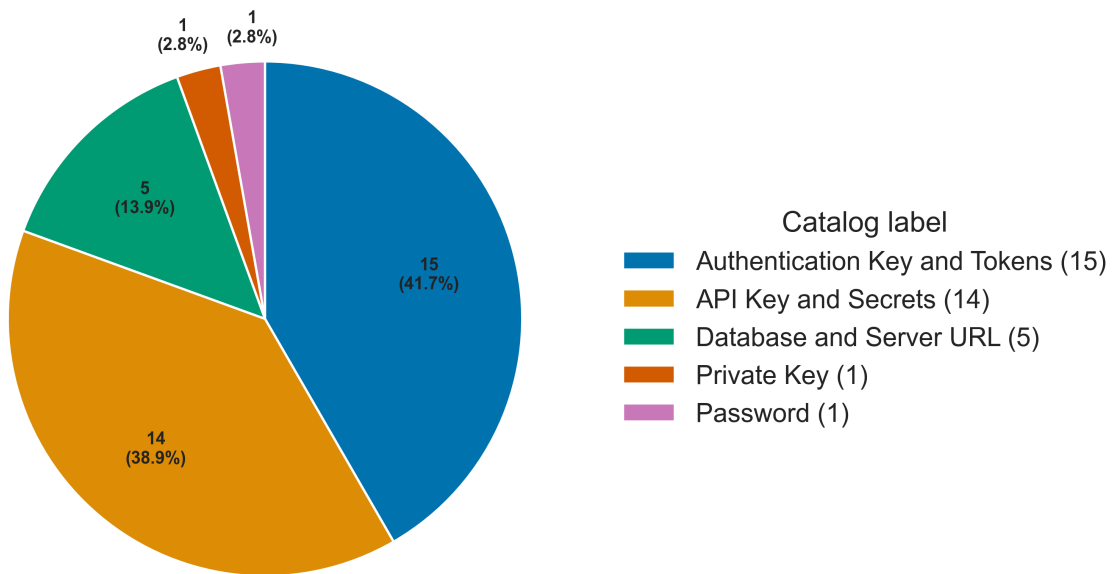


Figure 14: Distribution of the regex catalog.

6 Dataset and Experimental Setup

This chapter describes the dataset, the preprocessing decisions and the experimental setup used in the thesis. A synthetically generated dataset is used for controlled development, training, diagnostic analysis and the final reported evaluation of the models and the pipeline components. SecretBench [13], a benchmark derived from labeled secret occurrences in public GitHub repositories, was planned as an external benchmark, but access was not received in time and it is therefore not used for the reported results.

The chapter first defines the dataset requirements, then explains the unavailable SecretBench benchmark and the synthetic replacement, and documents the generated dataset, its exploratory analysis, split strategy and limitations. It then specifies how the baselines, the model-based stages and the final hybrid pipeline were executed and compared. The synthetic dataset enables reproducible implementation and controlled evaluation, while the missing SecretBench run defines the main external-validity limitation.

6.1 Dataset Requirements

The evaluation requires labeled text samples that represent software-artifact input and include either one or more annotated secrets or explicit negative examples without secrets. For every positive secret instance, the data must identify the sensitive value or its character span so that redaction can be evaluated at value level. The explicit labels for the marked secrets are not necessary for the performance evaluation, but mainly for the analysis of errors and to learn from them. The dataset must also support splitting into separate training, validation, and test partitions to be used during training of a component.

The two datasets fulfill different purposes in the experimental design. Table 6 summarizes this separation, which is maintained throughout the training and evaluation chapters.

Dataset	Origin	Role in this thesis	Main limitation
Synthetic dataset	Generated software artifacts and texts with non-functional secrets	Development, training, diagnostics, controlled internal evaluation and performance analysis	May contain generator-specific patterns and does not fully represent real-world complexity
SecretBench	Labeled secret candidates extracted from public GitHub repositories [13]	Planned external benchmark, but not used because access was not received in time	Represents public repository data (real secrets) and was not available during the main development phase

Table 6: Roles of the synthetic dataset and SecretBench in this thesis.

6.2 SecretBench and Access Delay

SecretBench [13] is a benchmark dataset derived from candidate secrets identified in public GitHub repositories using regex rules. According to its published description, the dataset contains 97,479 manually labeled candidate secrets extracted from 818 repositories, of which 15,084 are labeled as true secrets.

SecretBench is relevant for this thesis because it offers a real-world evaluation context that cannot be reproduced fully through synthetic generation. In particular, it contains the noise, formatting variation and non-secret candidates that arise in public software repositories. Especially when trying to reduce false-positives compared to existing tools using mainly regex rules and entropy, it provides a good challenge

during evaluation. For this reason, it was selected as the intended external benchmark for the final selected pipeline.

Category	True secrets	Total candidates
Private Key	5789	8584
API Key and Secrets	4529	5162
Authentication Key and Tokens	3569	5833
Other	524	66690
Generic Secret	334	439
Database and Server URL	162	9970
Password	150	705
Username	27	96
Total	15084	97479

Table 7: Published category distribution of SecretBench.

The published category distribution in [Table 7](#) also shows why SecretBench is valuable as an external evaluation dataset. The large number of candidate values labeled as non-secrets provides a realistic challenge for precision, while the presence of different true-secret categories supports analysis of whether the final pipeline generalizes well enough beyond the synthetic setting. However, because SecretBench is not used as the development dataset, its category distribution does not directly define the training balance or generation of the synthetic dataset.

Access to the labeled SecretBench dataset requires an approval process. The project requested access during the thesis period, but the dataset was not available during the main development and training phase. Consequently, SecretBench could not be used as the basis for model training, pipeline iteration or preliminary internal comparisons.

To keep the project executable within the available timeframe, the synthetic dataset was generated for development, training and controlled evaluation. Because SecretBench access was not received before the thesis had to be finalized, no final SecretBench run was performed. The pipeline-design and model-selection decisions are therefore based on the synthetic data only, and any later SecretBench experiment would be an external follow-up rather than a reported thesis result.

This distinction defines the dataset strategy of the thesis. SecretBench provides the intended external benchmark that could not be used, while the synthetic dataset provides controlled data for development, training, diagnostics and the reported evaluation. The following sections therefore analyze the synthetic dataset in detail.

6.3 Synthetic Dataset Generation

As a consequence of the unavailable SecretBench access described in [Section 6.2](#), a synthetic dataset generator was implemented. The resulting dataset serves as the controlled development and evaluation corpus of the thesis, used for model training, pipeline diagnostics, comparisons and the reported final benchmark. It is deliberately not treated as equivalent to real-world repository data.

The generated data is controlled through a generator configuration and deterministic logic based on a seed. The configuration defines contextual vocabularies, user and service names, password patterns and the selected generation policies. Procedural generation remains implemented in code for structured values whose syntax requires explicit construction, such as provider-specific tokens, private keys or credential-bearing connection strings. Because generation is seeded, the dataset is reproducible: re-running the generator with the same configuration yields the same artifacts and the same annotated spans.

A central design decision was to avoid reducing the dataset to simple assignments of known token formats. Passwords are intentionally included as a more challenging secret type because they may have comparatively low entropy and may only be identifiable from their surrounding context. In the final generator configuration, Password is configured as a hard secret type with a target ratio of 0.09.

The generator was refined iteratively during development. Early versions risked producing artifacts that were too simple or too closely aligned with rule-based detection. The current dataset therefore includes multiline artifacts, multiple secrets per positive artifact, broader artifact and language coverage, long connection strings, prose-like content and secret-like negative values. These design decisions are intended to create a controlled development dataset that exposes both recall and precision limitations of different detection approaches.

6.4 Dataset Format

The generated dataset is stored in JSON Lines format. Each row represents one complete software artifact or technical text fragment. A row contains the original input text, its assigned dataset split, a boolean indicator stating whether the artifact contains at least one annotated secret, a list of annotated secret spans and metadata describing the generated context including information from the generator. [Table 8](#) summarizes these fields.

Each secret span contains the synthetic secret value, its start and end position in the text, the generic model label SECRET and a more specific secret_type. The generic label is used because the primary task of the pipeline is redaction rather than provider-specific classification. If a sensitive value is removed from the output, the protection objective is achieved even if it is not assigned a specific category such as Password, APIKey or DatabaseURL. Nevertheless, the original secret types are preserved as metadata so that later analyses can investigate their distribution and which categories remain difficult to detect.

Level	Fields	Purpose
Artifact row	id, split, input, spans, contains_secret, metadata	Stores one complete generated software artifact or technical text fragment
Secret span	label, secret_type, value, start, end	Defines one annotated secret value and its exact character offsets
Metadata	artifact_type, language, file_type, template_id, template_family, document_shape, negative_types, source	Supports exploratory analysis, evaluation and category-level error analysis

Table 8: Structure of the generated JSONL dataset rows

Character offsets are stored directly during generation and validated before the dataset is used for training or evaluation. The validation checks whether each annotated value exactly corresponds to the substring at the stored character positions. In the analyzed snapshot, all 27,000 artifact rows and their annotated secret spans passed validation, and every span exactly matched its stored character offsets.

The generator can insert multiple secret values into one artifact. This avoids reducing the task to a one-secret-per-row shortcut and better resembles technical fragments in which several credentials may occur together, for example in a configuration file or deployment note.

For model training, all annotated secret spans are mapped to the single model-facing label Secret. The original generator-specific secret type is retained as metadata, but the span-detection task remains binary at the label level: a span either represents a secret value that should be redacted or it does not. The label description used for GLiNER2 narrows this task by defining Secret as real sensitive values such as API keys, access tokens, private keys, passwords, webhook secrets, client secrets, or credential-bearing

connection strings, while excluding ordinary identifiers, UUIDs, hashes, filenames, references, example values, and generic alphanumeric strings unless they clearly function as secrets.

6.5 Exploratory Data Analysis

Exploratory data analysis was performed to verify the generated dataset before it is used for model development and controlled evaluation. The analysis covers dataset size and class balance, secret-type and hard-negative coverage, artifact characteristics, length and entropy distributions, and possible exact or partial leakage between splits.

6.5.1 Dataset Composition

The dataset used in this thesis contains 27,000 generated artifacts, divided into the training, validation and test splits. The positive-artifact ratio is close to 65% in every split, which provides comparable conditions for later model and pipeline comparisons.

The training portion is further divided into `train_gliner` and `train_meta`. The GLiNER split is used for span-detector fine-tuning, while the meta split is reserved for generating candidate rows for the final LightGBM classifier. This avoids training the final candidate-level decision layer on the same cases used for GLiNER2 fine-tuning. Table 9 lists the split sizes and roles. The comparable positive ratios across the splits provide stable conditions for later model and pipeline comparisons.

Split	Artifacts	Positive ratio	Role	Final evaluation?
<code>train_gliner</code>	14,967	65.56%	GLiNER2 fine-tuning	No
<code>train_meta</code>	3,932	63.73%	Meta-classifier candidate training	No
<code>validation</code>	4,049	65.20%	Model selection and validation	No
<code>test</code>	4,052	65.05%	Synthetic test set	Yes

Table 9: Synthetic dataset split sizes and roles.

6.5.2 Secret-Type Coverage

Although training uses the single generic label SECRET, each generated span retains its original secret type as metadata. The current dataset contains 53 distinct secret types. This enables type-specific evaluation without changing the main training and redaction task into a multi-class classification problem.

The most frequent secret type is Password, with 4,107 annotated spans, corresponding to 10.96% of all target spans. This overrepresentation is intentional, as passwords are particularly relevant to the thesis. They may be short, low-entropy, and context-dependent, which makes them difficult to detect using only regular expressions or entropy-based methods. The next most frequent categories include GenericSecret (3.76%), SessionCookie (3.61%), APIKey (3.60%) and WebhookSigningSecret (3.45%). In addition, the dataset contains long structured values such as storage connection strings, database URLs, Redis URLs, SMTP URLs, and connection strings. Figure 15 shows the full distribution by secret type.

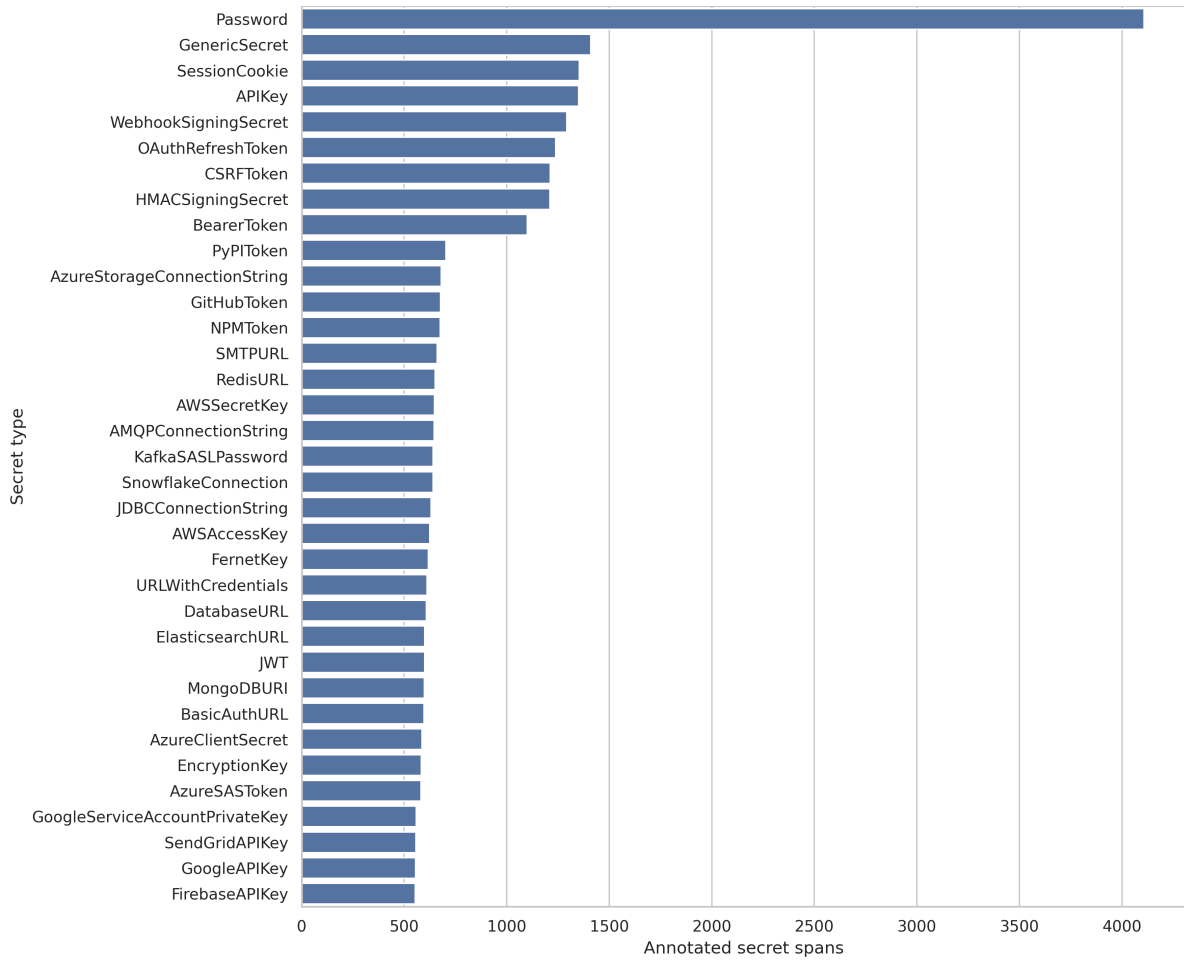


Figure 15: Distribution of annotated secret spans by original secret type.

The variety of retained secret categories supports two purposes. First, it ensures that model development is not based solely on easily recognizable provider-prefix formats. Second, it allows the later evaluation to determine whether the developed pipeline performs differently on short contextual secrets, structured tokens and long compound secret values.

6.5.3 Artifact and Secret Characteristics

The dataset generator creates artifacts with varying lengths, line counts and numbers of embedded secrets. This is relevant because the target use case includes copied texts such as code fragments, configuration files, logs and messages rather than only isolated token values.

The current dataset snapshot also records the share of secret characters across all artifact characters, which is 16.79%. This is useful for interpreting exact-span metrics: a small number of long connection strings or private-key blocks can represent a large number of secret characters even when they correspond to only a few spans. [Figure 16](#) shows the artifact and secret length and entropy distributions.

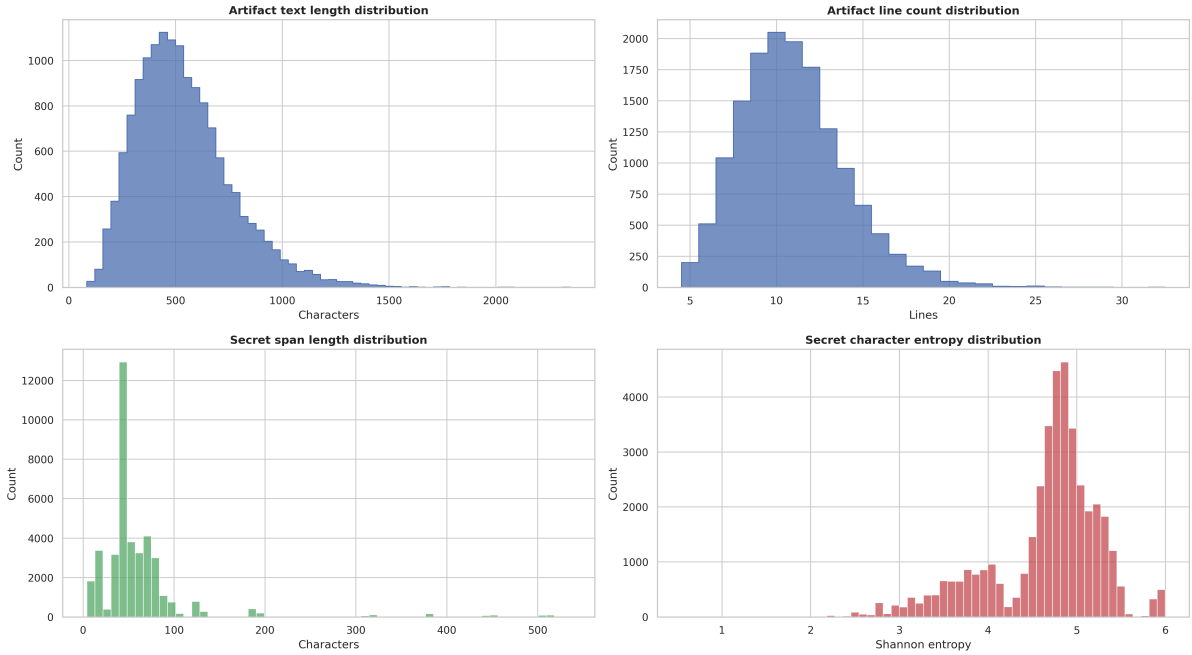


Figure 16: Length and entropy distributions of generated artifacts and secret spans.

These characteristics are relevant for interpreting later results. Failure cases on long strings may relate to span-boundary detection, whereas failures on short or low-entropy secrets may indicate the need for contextual model-based detection.

6.6 Negative Examples

A practical secret-redaction system must distinguish actual credentials from values that merely resemble secrets. The generator therefore inserts hard-negative values into both positive and negative examples. These values are intentionally selected because they may be flagged by broad regex rules, entropy-based heuristics or neural detectors despite not representing functional credentials.

The current dataset contains 37 distinct hard-negative types. The most frequent hard-negative categories are sample connection strings (4,028 occurrences, 7.08% of inserted negative candidates), sample passwords (4,022, 7.07%), sample provider tokens (3,894, 6.84%) and sample JSON Web Tokens (3,878, 6.81%). Further negative types include public certificates, placeholder values, public URLs, UUIDs, request identifiers, hashes, environment-variable references and intentionally masked or empty values.

The inclusion of hard negatives is important for the design of the evaluation. A pipeline that classifies every secret-shaped value as sensitive may achieve high recall, but would redact technical information unnecessarily. Hard negatives therefore make it possible to analyze the precision trade-off of broad detection strategies and later candidate-classification stages.

The negative examples are also reflected in the GLiNER2 label description. The model is explicitly instructed not to treat ordinary identifiers, UUIDs, hashes, filenames, references, example values, or generic alphanumeric strings as secrets unless they clearly function as actual credentials. This mirrors the purpose of the hard-negative examples: the model and the later classifier should learn that secret-like appearance alone is not sufficient for redaction.

6.7 De-duplication and Leakage Prevention

Synthetic generation enables exact annotations and controlled dataset construction, but it also creates a risk of overly similar artifacts across development and evaluation splits. A model could achieve overly optimistic results if identical secret values, identical full texts or recurring template structures appear across splits. The dataset was therefore validated at row level, dataset level and split level.

At row level, the validation checks the required schema fields, the consistency of `contains_secret` with the annotated spans, the validity of character offsets and the absence of invalid overlapping spans.

At dataset level, identifiers and complete input texts are checked for duplicates. In the analyzed snapshot, no duplicate complete inputs and no duplicate exact secret values were detected.

At split level, the validation checks whether identical input texts or identical secret values occur across different splits. The analyzed dataset contains no secret values repeated across splits, no exact input texts repeated across splits and no exact normalized values repeated across splits.

Exact duplicate checks do not completely eliminate the risk of structural similarity. For example, two generated configuration files may use different generated credentials while retaining similar surrounding syntax and layout. To investigate this remaining risk, normalized token-set Jaccard similarity [32] is analyzed across splits. Before comparison, annotated secrets and hard-negative values are replaced with placeholders so that the analysis focuses on surrounding artifact structure rather than on the secret values themselves (Figure 17).

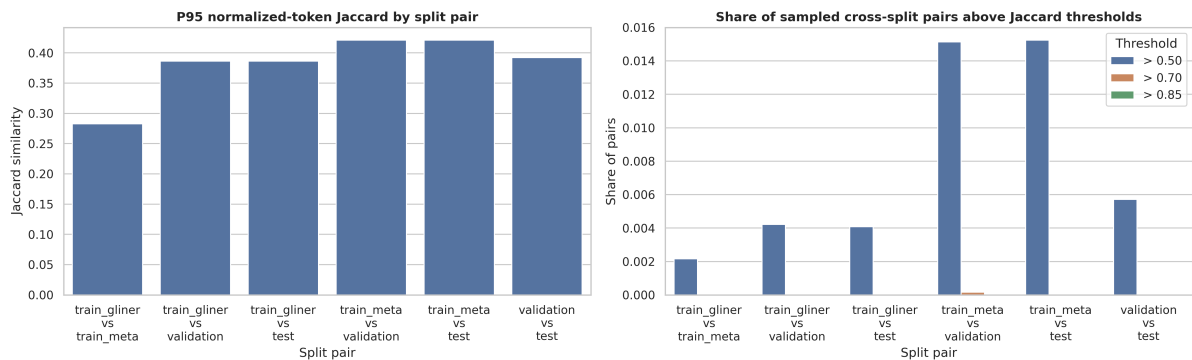


Figure 17: Normalized token-set Jaccard similarity across dataset splits.

In random cross-split sampling, the maximum observed Jaccard similarity was 0.7143, and the largest 95th-percentile similarity across split pairs was 0.3922. The largest sampled share of pairs at or above the diagnostic threshold of 0.70 was 0.006%, indicating that highly similar structures are rare in random cross-split comparisons. Nevertheless, a targeted scan identified 70 high-similarity pairs, with a maximum similarity of 0.875. This confirms that isolated structurally similar artifacts remain present and must be treated as a limitation of the synthetic dataset rather than ignored.

6.8 Train, Validation and Test Set

The generator splits the development dataset using configured ratios of 70% for training, 15% for validation and 15% for the test set. The split is designed to keep the distributions of template families, artifact types and positive or negative artifacts comparable across the three sets, which supports stable model development and interpretable validation and test comparisons.

The training portion is split again into `train_gliner` and `train_meta`. The meta split contains 3,932 rows, about 20.8% of the training portion. Assignment to the two splits is grouped by `template_id`, with

an exact `template_id` overlap of 0. This separation matters because the final LightGBM meta-classifier is trained on candidate rows generated from `train_meta`, not from the cases used to fine-tune GLiNER2.

6.9 Dataset Limitations

The synthetic dataset provides a controlled basis for development. It is reproducible, avoids using real credentials, contains exact-matching target spans and allows explicit control over secret types, artifact contexts and difficult negative examples.

However, synthetic data cannot fully represent the diversity and irregularity of real software repositories, logs and developer workflows. Although the generator includes multiple artifact families and secret categories, its examples are still produced from predefined templates and policies. A model or pipeline may therefore perform well partly because it learns regularities of the generator rather than properties that generalize fully to real-world artifacts.

The similarity analysis reduces the risk of unnoticed leakage by checking exact overlap and quantifying structural similarity between splits. Nevertheless, the Jaccard scan shows that isolated structurally similar artifacts remain possible and must be treated as a documented limitation.

A further limitation concerns span policy. The synthetic dataset labels the complete generated secret value. For compound values such as connection strings or credential-bearing URLs, this may correspond to a longer span than an external dataset would annotate. Exact-matching evaluation can therefore penalize detections that mask a sensitive substring but do not cover the complete generator-defined span.

For these reasons, synthetic evaluation results are interpreted as controlled evidence for development rather than as a definitive estimate of real-world performance. A future SecretBench run or another labeled real-world dataset would be required to externally validate the final selected pipeline.

6.10 Experimental Setup

This part of the chapter specifies how the baselines, the model-based stages and the final hybrid pipeline were executed and compared. It documents the experiments that were actually run and for which result artifacts are retained.

6.11 Baseline Implementations

The non-neural baseline experiment evaluates the deterministic candidate-generation stages independently from the later model-based components. The evaluation is limited to candidate spans emitted by regex and entropy-based detection stages.

All runs use the secret-only evaluation mode, so any annotated secret span is treated as SECRET, independently of its fine-grained category. This prevents label differences from influencing the comparison of candidate generators and is consistent with the redaction objective defined in [Section 4.3](#). [Table 10](#) summarizes the baseline configuration.

Aspect	Configuration
Evaluation set	test from the synthetic dataset described in Section 6.8 .
Primary matching	Exact value-level span matching as defined in Section 4.4
Secondary matching	Partial matching, including partial-only rate and the average overlap
Regex baseline	Using the 36-pattern catalog described in Section 5.4
Entropy baseline	Minimum candidate length 16 and thresholds 3.0, 3.5, 4.0, and 4.5
Combined baseline	The union of regex and entropy detections at the same tested entropy thresholds

Table 10: Configuration of the regex and entropy baseline experiments

6.12 Model Comparison Setup

The model comparison supports the selection of the neural span-detection component for the hybrid pipeline. It is not intended as an exhaustive benchmark of all possible secret-detection architectures. Instead, it compares a small set of candidate models that can be used as local span detectors or related secret-detection baselines before task-specific fine-tuning.

All compared models are evaluated in a zero-shot setting with the same model-facing Secret label description. The comparison therefore answers a narrow question: which base model is the most promising starting point for later fine-tuning and local integration? The result is not used as the final pipeline score. Final quality is measured later with the fine-tuned GLiNER2 component and the complete hybrid pipeline. [Table 11](#) lists the compared models and their roles.

Model group	Representative model	Pipeline role	Reason for inclusion
OpenAI Privacy Filter [33]	openai/privacy-filter	General privacy-oriented span detection	Strong privacy baseline with secret-like detections, but not used as the local model to refine
GLiNER2	fastino/gliner2-base-v1, fastino/gliner2-large-v1	Flexible span detection	Initial thesis direction and direct support for descriptive labels and fine-tuning
GLiNER v2.1	urchade/gliner_small-v2.1, urchade/gliner_medium-v2.1, urchade/gliner_large-v2.1	Flexible span detection	Predecessor architecture family and size comparison.
GLiNER-PII	nvidia/gliner-PII	PII-oriented span detection	Checks whether a PII-oriented GLiNER model transfers to software secrets.
DeepPass2 [34]	gneeraj/deepass2-bert	Password-oriented token classification	Domain-related baseline for password and credential-like values.

Table 11: Scope of the zero-shot model comparison

The comparison uses precision, recall, F1 and F2. Runtime per example is also considered because the selected model must later run inside a local desktop prototype. The comparison is therefore interpreted as a quality-runtime trade-off rather than as a pure leaderboard.

We evaluate at least the five configurations listed in Table 12. The number of parameters is rounded for readability, the exact figures are taken from the model cards.

Model	Parameters	Pretraining	Reason for inclusion
GLiNER small [8]	50M	Multi-domain NER	Smallest variant, CPU-friendly baseline.
GLiNER medium [8]	200M	Multi-domain NER	Strongest balance between size and quality.
GLiNER large [8]	450M	Multi-domain NER	Reference for the best CPU-feasible variant.
GLiNER-PII [35]	200M	PII-focused fine-tune	Closest existing baseline for sensitive entities.
GLiNER2 [9]	205M	Multi-task schema	Multi-task interface, strong recent baseline.

Table 12: Pretrained models compared in zero-shot mode.

6.13 GLiNER2 LoRA Fine-Tuning Setup

The model-based span detector is fine-tuned as a single-label GLiNER2 configuration. Fine-grained secret types such as Password, APIKey, or DatabaseURL remain available as metadata for analysis, but they are not used as separate model labels and are collapsed to Secret.

The fine-tuning run uses the base model fastino/gliner2-base-v1. A LoRA adapter is trained instead of fully fine-tuning all model parameters. LoRA is used to keep the training resource requirements manageable while still adapting the model to the synthetic secret-redaction domain. The trainer configuration is summarised in Table 13.

Parameter	Value
Base model	fastino/gliner2-base-v1
Training epochs	10
Batch size	4
Gradient accumulation	4
Encoder learning rate	5e-6
Task learning rate	2e-4
Scheduler	cosine
Warmup ratio	0.08
Weight decay	0.02
Max. gradient norm	1.0
LoRA rank	16
LoRA alpha	32.0
LoRA dropout	0.1
Early stopping patience	3

Table 13: Fine-tuning configuration of the GLiNER2 LoRA adapter.

Before the final GLiNER2 adapter was trained for use in the complete redaction pipeline, an additional set of controlled LoRA configuration experiments was executed. These experiments were designed as a hyperparameter and robustness study. They compare different epoch counts, learning rates and LoRA capacities for the targeted task. The results are therefore used to guide and justify the training setup, but they should not be confused with the final end-to-end pipeline evaluation reported later in the thesis.

The configuration experiments use the same base model, fastino/gliner2-base-v1, and the same single model-facing label Secret. Evaluation is performed with exact character-span matching at a prediction threshold of 0.80. Precision, recall, F1 and F2 are reported, with F2 included because missed secrets are more problematic than limited over-redaction in the redaction scenario.

The final full-data training run in [Table 13](#) intentionally differs from the best configuration found in these experiments. The experiments were conducted on small subsets to keep runtime manageable, and their primary purpose was to understand the rough sensitivity of the model to different hyperparameter choices and to identify configurations that are clearly insufficient. They established that a smaller adapter is not harmful, that recall is the primary differentiator between configurations, and that early stopping is appropriate because the model learns the main synthetic patterns quickly. The final training run then applies these insights on the full dataset, but uses a more conservative configuration (LoRA rank 16, alpha 32, dropout 0.1 during 10 epochs) to ensure sufficient adapter capacity when training on the considerably larger full dataset rather than the subsets used here. A configuration that performs well on a small sample does not necessarily transfer directly to training with more data. The experiments should therefore be understood as a source of direction and an exploration of the limits of the approach, not as a direct optimisation of the final adapter.

6.13.1 Experiment Subsets

The configuration experiments were not run on the full training dataset. Instead, stratified synthetic subsets were used to keep the runtime manageable while still allowing a controlled comparison between configurations. This means that the results in this subsection are approximate training-study results, not the final full-data result of the thesis.

Split	Rows	Positive rate
train	2,500	68.44%
validation	800	77.38%
test	800	76.63%

Table 14: Synthetic subsets used for the GLiNER2 LoRA configuration experiments.

The high positive rates in [Table 14](#) are useful for comparing span-detection behaviour on secret-heavy samples, but they also limit how strongly the absolute values can be generalized. For this reason, the configuration experiments are interpreted as model-selection evidence and not as a replacement for the later complete pipeline evaluation.

Eight configurations were completed and are shown in [Table 15](#). The baseline configuration uses rank 16, alpha 32, dropout 0.10 and six epochs. The runs adjust one main factor at a time, either epoch count, learning rate, or LoRA capacity/dropout.

Run	Epochs	Encoder LR	Task LR	LoRA r	Alpha	Dropout
baseline	6	5e-6	2e-4	16	32	0.10
4_epochs	4	5e-6	2e-4	16	32	0.10
10_epochs	10	5e-6	2e-4	16	32	0.10
1e4_lr	6	2.5e-6	1e-4	16	32	0.10
5e4_lr	6	1e-5	5e-4	16	32	0.10
lora_r8_a16_d05	6	5e-6	2e-4	8	16	0.05
lora_r32_a64_d15	6	5e-6	2e-4	32	64	0.15

Table 15: Executed GLiNER2 LoRA configuration experiments.

6.13.2 Experiment Results

The configuration experiment results sorted by test F2 are shown in Table 16. The best configuration is lora_r8_a16_d05, which uses a smaller LoRA adapter with rank 8, alpha 16, dropout 0.05 and six training epochs.

Run	Precision	Recall	F1	F2
lora_r8_a16_d05	0.9814	0.7928	0.8771	0.8245
5e4_lr	0.9760	0.7878	0.8719	0.8194
10_epochs	0.9882	0.7814	0.8727	0.8155
lora_r32_a64_d15	0.9900	0.7806	0.8729	0.8151
baseline	0.9820	0.7806	0.8698	0.8140
4_epochs	0.9819	0.7792	0.8689	0.8128
1e4_lr	0.9813	0.7513	0.8510	0.7882

Table 16: Exact-span GLiNER2 LoRA configuration experiment results sorted by test F2.

Compared with the baseline configuration, the best configuration improves test F1 from 0.8698 to 0.8771 and test F2 from 0.8140 to 0.8245. The improvement is mainly caused by higher recall, which increases from 0.7806 to 0.7928, while precision remains almost unchanged. This pattern is important because the thesis prioritizes missed-secret reduction over maximum precision.

The result does not indicate a dramatic change in model behaviour. Instead, it suggests that a smaller adapter is sufficient for this synthetic secret-redaction task and may generalize slightly better on the sampled test subset. The configuration experiments should therefore be interpreted as a hyperparameter preference rather than as a standalone proof that the smaller adapter is universally superior.

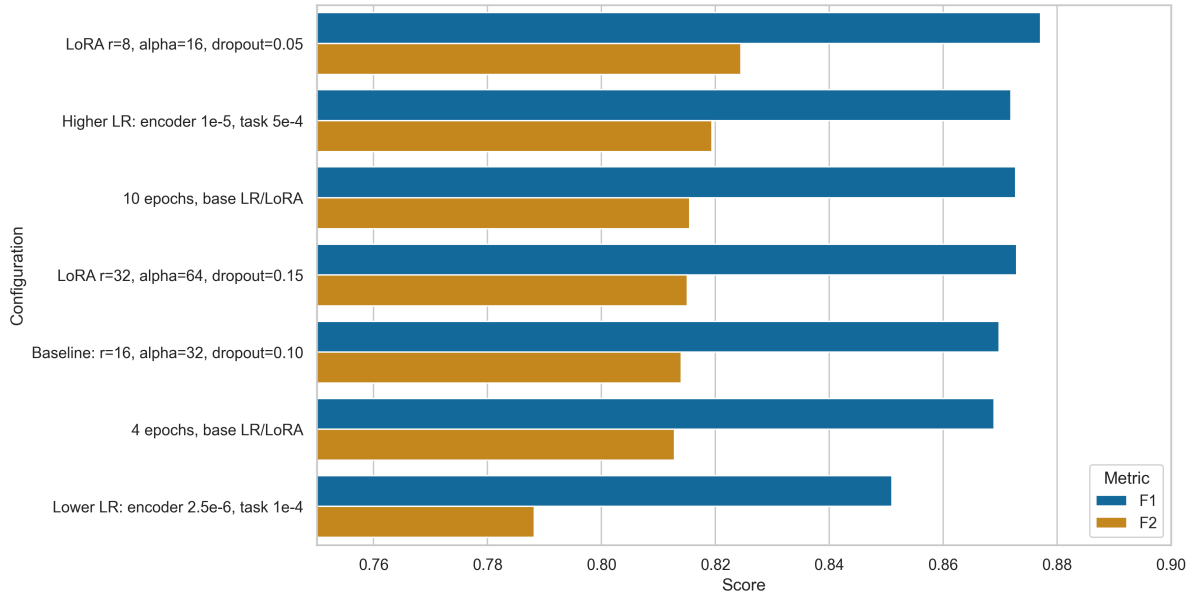


Figure 18: F1 and F2 of performed GLiNER2 LoRA configuration experiments.

6.13.3 Five-Fold Robustness Check

A complete 5-fold cross-validation was not run for every configuration because this would multiply the training cost. Instead, the best configuration `lora_r8_a16_d05` was evaluated with five folds as a robustness check and the mean and standard deviation are shown in Table 17.

Metric	Mean	Std. Deviation
Validation precision	0.9895	0.0067
Validation recall	0.7435	0.0464
Validation F1	0.8485	0.0321
Validation F2	0.7821	0.0417
Test precision	0.9897	0.0015
Test recall	0.7894	0.0048
Test F1	0.8783	0.0032
Test F2	0.8227	0.0043

Table 17: Five-fold cross-validation summary of best GLiNER2 LoRA configuration.

The test metrics are stable across the five folds. Test F2 has a mean of 0.8227 and a standard deviation of only 0.0043, while test F1 has a mean of 0.8783 and a standard deviation of 0.0032.

The validation metrics are considerably less stable, as reflected in the high standard deviations for the validation metrics compared with their test counterparts. The cause is the uneven fold composition shown in Table 18, where validation fold sizes range from 1,442 to 295 rows and positive rates range from 83.91% down to 30.85%.

Validation split	Rows	Positive rate
fold 1	1,442	83.91%
fold 2	755	70.33%
fold 3	472	53.39%
fold 4	336	38.10%
fold 5	295	30.85%

Table 18: Validation fold balance for the five-fold robustness check.

This imbalance is a direct consequence of the splitting strategy used when constructing the cross-validation splits. The split logic grouped samples by multiple metadata attributes simultaneously. It checked whether the sample contains a secret, the type of template in the generator, and the secret type. Some of these combined groups are very small while others are much larger. When the resulting groups were distributed across five folds, the algorithm preserved these metadata distinctions at the expense of fold-size balance. Consequently, some validation folds received substantially more samples and a much higher proportion of positive examples than others.

The 5-fold cross-validation should therefore be interpreted as a supplementary robustness check rather than a balanced cross-validation experiment. The stable test metrics are still informative, they confirm that the model’s generalisation does not depend on a single favourable data split, but the validation metrics carry a structural caveat and should not be compared directly across folds or used in isolation as performance estimates.

6.13.4 Interpretation

The experiments support three conclusions. First, all tested GLiNER2 LoRA configurations achieve high precision on the sampled synthetic test subset, which means that false positives are not the main difference between the runs. Second, recall is the more important source of variation, and the best configuration `lora_r8_a16_d05` improves F2 mainly by improving recall. Third, longer training or a larger adapter does not automatically improve exact-span performance. A smaller adapter with lower dropout is sufficient for this task and performs slightly better on the sampled subset.

These findings are consistent with the broader thesis argument that the neural detector should be treated as one component in a larger hybrid system. The GLiNER2 model contributes context-sensitive candidate spans, but the final redaction quality depends on how those candidates are combined with regex, entropy and classifier-based filtering. The configuration experiments and CV results therefore justify the feasibility and stability of LoRA-based GLiNER2 adaptation, while the final pipeline evaluation remains the decisive evidence for the complete redaction approach.

6.13.5 Limitations of the Training Experiments

The training experiments have several limitations. The configuration experiments are based on sampled synthetic subsets rather than the full dataset. The sampled validation and test subsets also have high positive rates, so the absolute metric values should not be interpreted as production performance. The cross-validation folds are also structurally imbalanced, as described in [Section 6.13.3](#). Finally, the experiments evaluate exact character-span matching only and do not include partial-match metrics.

6.14 Meta-Classifer Training

The final decision layer is trained as a supervised candidate-level binary classifier. It receives candidate spans that were produced by the earlier pipeline stages and predicts whether each candidate should be retained as a true Secret or rejected as No Secret. The meta-classifier is therefore a post-detection component: it does not extract spans from raw text by itself, but combines evidence from regex, entropy, GLiNER2, optional Secrets-Sentinel classification, candidate-shape features and local context.

The training target is binary. A candidate is positive if it corresponds to an annotated secret span and negative if it corresponds to a hard negative or another non-secret candidate produced by the pipeline. Candidate rows are split by `case_id` into train, validation and test partitions. This grouped split avoids candidate-level leakage where multiple candidates from the same generated artifact would otherwise occur in different splits.

A reduced feature set is used for the thesis and deployment export. The raw `candidate_text` feature is intentionally disabled because it could encourage memorization of generator-specific strings and would make the model harder to justify analytically. Instead, the feature schema is limited to interpretable signals that correspond to the previous pipeline stages or to simple candidate properties.

Feature family	Examples	Purpose
Stage provenance	<code>matched_by_regex</code> , <code>matched_by_entropy</code> , <code>matched_by_gliner_entity</code> , <code>matched_stage_count</code> , <code>candidate_stage</code> , <code>regex_pattern_name</code>	Records which upstream stages support the candidate and whether several stages agree.
Entropy signals	<code>candidate_entropy</code> , <code>best_entropy_score</code> , <code>best_entropy_value</code>	Represents the direct randomness of the candidate and entropy-stage support.
Stage and model scores	<code>candidate_score</code> , <code>best_regex_score</code> , <code>best_gliner_entity_score</code> , optionally <code>classification_score</code>	Preserves confidence information from upstream stages and the optional Secrets-Sentinel signal.
Candidate shape	<code>candidate_length</code> , <code>candidate_digit_ratio</code> , <code>candidate_special_ratio</code> , <code>candidate_has_dot</code> , <code>candidate_is_url_like</code> , <code>candidate_is_pem_like</code>	Captures structural properties that distinguish many credentials from ordinary strings.
Context signals	<code>context_length</code> , <code>context_has_assignment</code> , <code>context_has_secret_keyword</code>	Uses simple local-context indicators without relying on raw text memorization.

Table 19: Feature families used for candidate-level meta-classifier training.

Two variants are trained. The first variant includes the `classification_score` produced by the optional Secrets-Sentinel stage. The second variant excludes this feature and is therefore deployable without running Secrets-Sentinel. Both variants use a fixed reduced feature set with 14 numeric or Boolean features and 2 categorical features (Table 20).

Variant	Includes classification_score	Numeric/ Boolean features	Categorical features	with classifier feature
Yes	14	2	without classifier feature	No
14	2			

Table 20: Meta-classifier variants and their feature availability assumptions.

Only LightGBM [31] is used for the final pipeline because it provides strong performance, fast inference and interpretable feature-importance outputs for the reduced feature schema. Earlier script versions supported additional classical model families, but the final exported comparison focuses on the selected model. The training configuration uses feature pruning with a maximum of 14 numeric/Boolean features per variant, keeps the two categorical features, and disables the raw candidate-text feature.

6.15 Hybrid Ablation Setup

The hybrid benchmark evaluates the complete secret-redaction pipeline as an end-to-end detection system. The goal is to measure how the final pipeline behaves when later decision layers are added to the same upstream candidate-generation stages. All four reported runs shown in Table 21 use the same benchmark file, the synthetic test.jsonl split, and the same upstream detector family: regex, entropy and GLiNER2 entity detection. The runs differ only in whether Secrets-Sentinel and the LightGBM meta-classifier are enabled.

Run	Regex	Entropy	GLiNER2	Secrets-Sentinel	Meta-classifier
No classifier	Yes	Yes	Yes	No	No
Secrets-Sentinel	Yes	Yes	Yes	Yes	No
Meta + Sentinel	Yes	Yes	Yes	Yes	Yes
Meta only	Yes	Yes	Yes	No	Yes

Table 21: Hybrid pipeline benchmark configurations.

The benchmark reports exact and partial matching. Exact matching is the primary metric because the predicted span must match the target span boundaries exactly. Partial matching is reported as a secondary diagnostic because it shows how often a secret is at least covered by an overlapping prediction. In the redaction use case, partial detection may still remove sensitive text, but exact matching is the stricter and more reproducible measure. The shared benchmark settings are listed in Table 22.

Parameter	Value
Benchmark file	synthetic_secrets/test.jsonl
Cases	4052
Target secret spans	6047
Regex base score	0.92
Entropy minimum length	16
Entropy threshold	3.5
GLiNER entity confidence	0.85
Context window	120 characters left and right
De-duplication strategy	highest_score
Redaction mode	label_token

Table 22: Shared benchmark settings for the final hybrid pipeline comparison.

6.16 Threats to Reproducibility

Several aspects can make a fine-tuning run hard to reproduce. We document the mitigations explicitly so that a reader can replicate the experiments.

- **Data version.** The synthetic generator configuration, generated data snapshot and split files are versioned. The exported splits are stored with SHA-256 fingerprints reported in the appendix.
- **Software version.** `requirements.txt` and `Cargo.lock` pin every dependency. The Tauri build script verifies the SHA-256 of every downloaded model artifact.
- **Random seeds.** All scripts set deterministic seeds for the data shuffler, the model initialization and the sampler.
- **Hardware.** All experiments were conducted on a Kaggle-hosted [\[36\]](#) cloud environment. The system was equipped with an NVIDIA Tesla P100 GPU, an Intel Xeon CPU running at 2.00 GHz with 4 virtual CPUs, and 29 GB of RAM.

7 Results

This chapter reports the measured experimental results on the synthetic test set defined in [Section 6.8](#). Exact value-level matching is the primary criterion, and partial matching is reported as a secondary analysis according to [Section 4.4](#). The regex and entropy baselines are reported in [Section 7.1](#), the fine-tuned GLiNER2 model in [Section 7.3](#), the candidate-level LightGBM meta-classifier in [Section 7.5](#), and the end-to-end hybrid pipeline in [Section 7.6](#). F2 is the primary aggregate metric throughout, while F1, precision and recall are also reported.

7.1 Baseline Performance

The first experiment measures the two non-neural candidate-generation mechanisms: the regex stage and Shannon-entropy-based detection. Their union is also evaluated to determine whether entropy contributes coverage beyond recognizable regex formats.

[Table 23](#) reports exact value-level results for the regex-only baseline and the strongest entropy-containing variant among the evaluated entropy thresholds ($t = 4.5$). The complete threshold experiments are analyzed in [Section 7.8.5](#).

Configuration	Precision	Recall	F1	F2	Runtime (ms)
Regex only	0.6132	0.6180	0.6156	0.6170	1
Entropy only, $t = 4.5$	0.3152	0.3868	0.3473	0.3700	2
Regex + entropy, $t = 4.5$	0.5113	0.7631	0.6123	0.6947	3

Table 23: Exact value-level results of the regex and entropy baselines.

Under exact matching, regex-only achieves the highest F1 (0.6156). Adding entropy at threshold 4.5 raises recall from 0.6180 to 0.7631, but the precision drop from 0.6132 to 0.5113 leaves F1 just below the regex-only result (0.6123). Under F2, however, the combined variant is stronger (0.6947 against 0.6170) because F2 rewards the recall increase. A collection of these results is shown in [Figure 19](#).

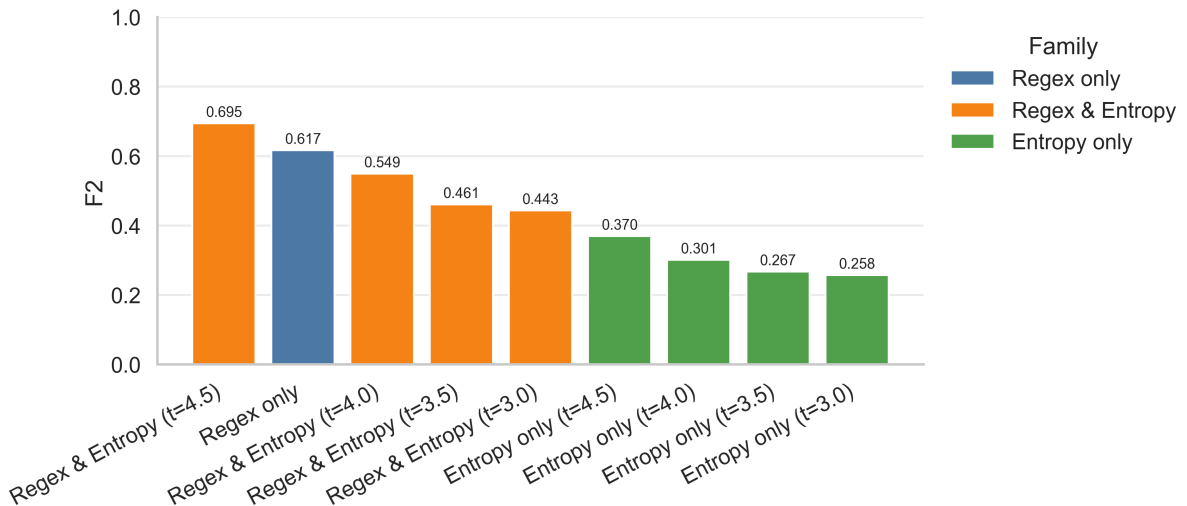


Figure 19: Exact-matching F2 of regex-only, entropy-only and combined variants.

Partial matching provides an additional view, because entropy detections may cover most of a long secret without reproducing its full annotated length. [Table 24](#) reports the same configurations under the secondary partial-matching criterion.

Configuration	Precision	Recall	F1	F2	Partial recovery	Avg. overlap
Regex only	0.6467	0.6518	0.6492	0.6508	0.0885	85.63%
Entropy only, $t = 4.5$	0.5281	0.6481	0.5820	0.6199	0.4261	82.43%
Regex + entropy, $t = 4.5$	0.5860	0.8746	0.7018	0.7962	0.4706	95.64%

Table 24: Partial-matching results of the regex and entropy baselines.

Under partial matching shown in Figure 20, the combined regex-entropy variant at threshold 4.5 achieves the highest partial-matching F1 (0.7018) and F2 (0.7962). It recovers 47.06% of the secrets missed by exact matching, with an average overlap of 95.64%, which indicates that entropy adds near-complete detections rather than only boundary touches.

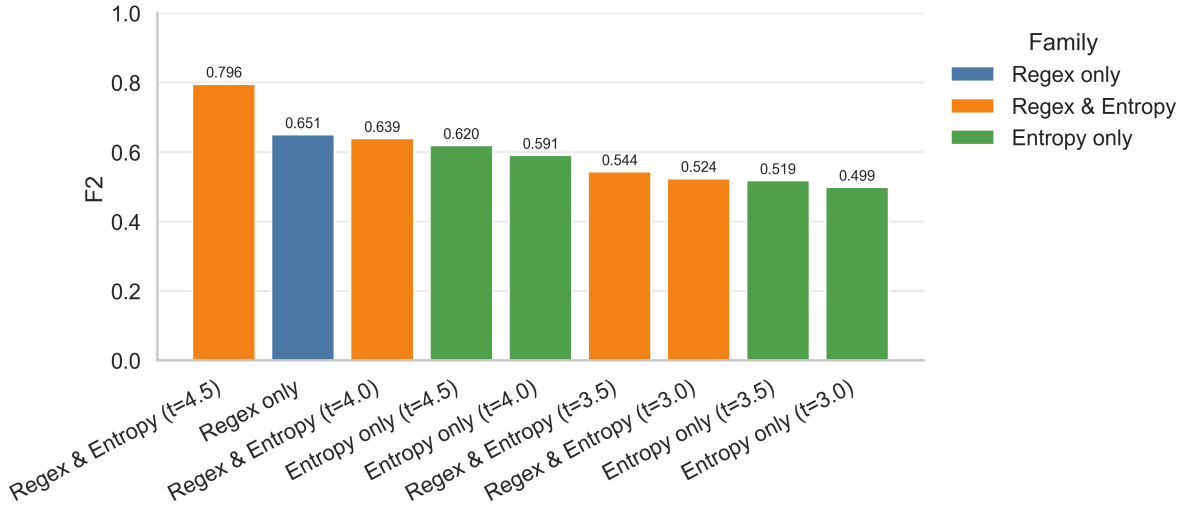


Figure 20: Partial-matching F2 of regex-only, entropy-only and combined variants.

Taken together, the baselines support two conclusions. First, a carefully selected regex catalog is a strong non-neural baseline for exact redaction on this synthetic dataset. Second, entropy is useful as an additional high-recall candidate signal, especially under recall-weighted metrics such as F2, but its false-positive behavior makes it unsuitable as a standalone redaction decision or as an unfiltered union with regex. This motivates the later hybrid stages, where entropy generates candidates and a learned decision layer filters them.

7.2 Model Comparison

The zero-shot model comparison is used to decide which model family should be fine-tuned for the redaction pipeline. The models are evaluated as span detectors before task-specific fine-tuning. This means that the absolute scores are expected to be lower than the later fine-tuned and hybrid-pipeline results. The comparison is still useful, it shows whether the selected base model is a reasonable starting point and whether a larger or externally trained alternative provides enough benefit to justify additional runtime and integration effort.

Table 25 reports the best exact-matching result per model. The strongest exact-matching F2 is achieved by the OpenAI Privacy Filter [33] with F2 0.5231. This is substantially higher than the GLiNER and DeepPass2 zero-shot results, but it comes with a very large runtime cost: 1972.39 ms per example compared with 31.87 ms for GLiNER2 base. Among the GLiNER-family models, GLiNER2 base remains the strongest exact-matching-F2 option and is much faster than GLiNER2 large. Figure 21 and Figure 22 visualize exactly this trade-off.

Model	Precision	Recall	F1	F2	TP	FP	Runtime
Privacy Filter	0.3250	0.6172	0.4258	0.5231	3797	7886	1972.39 ms
GLiNER2 base	0.2277	0.2227	0.2251	0.2237	1370	4648	31.87 ms
GLiNER-P11	0.4476	0.1977	0.2742	0.2225	1216	1501	72.28 ms
GLiNER2 large	0.3897	0.2007	0.2650	0.2223	1235	1934	73.94 ms
DeepPass2	0.1330	0.2381	0.1707	0.2056	1465	9546	30.10 ms
GLiNER large	0.1475	0.0770	0.1012	0.0852	474	2739	72.60 ms
GLiNER medium	0.1564	0.0226	0.0395	0.0273	139	750	35.36 ms
GLiNER small	0.0641	0.0037	0.0071	0.0046	23	336	23.90 ms

Table 25: Zero-shot model comparison under exact matching.

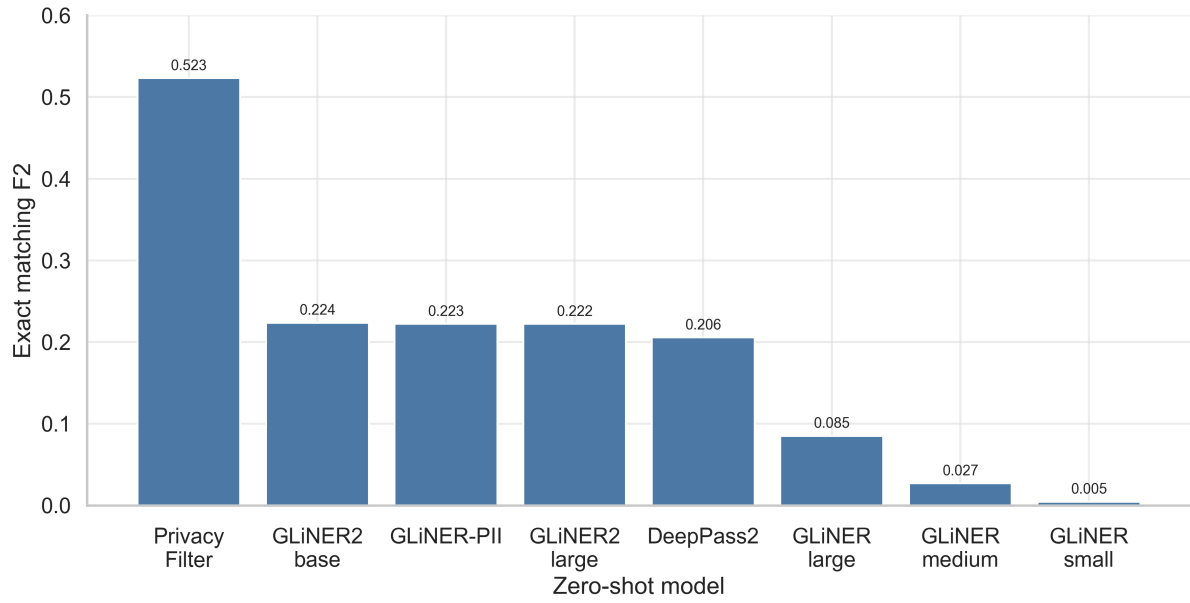


Figure 21: Exact matching F2 of the zero-shot model comparison.

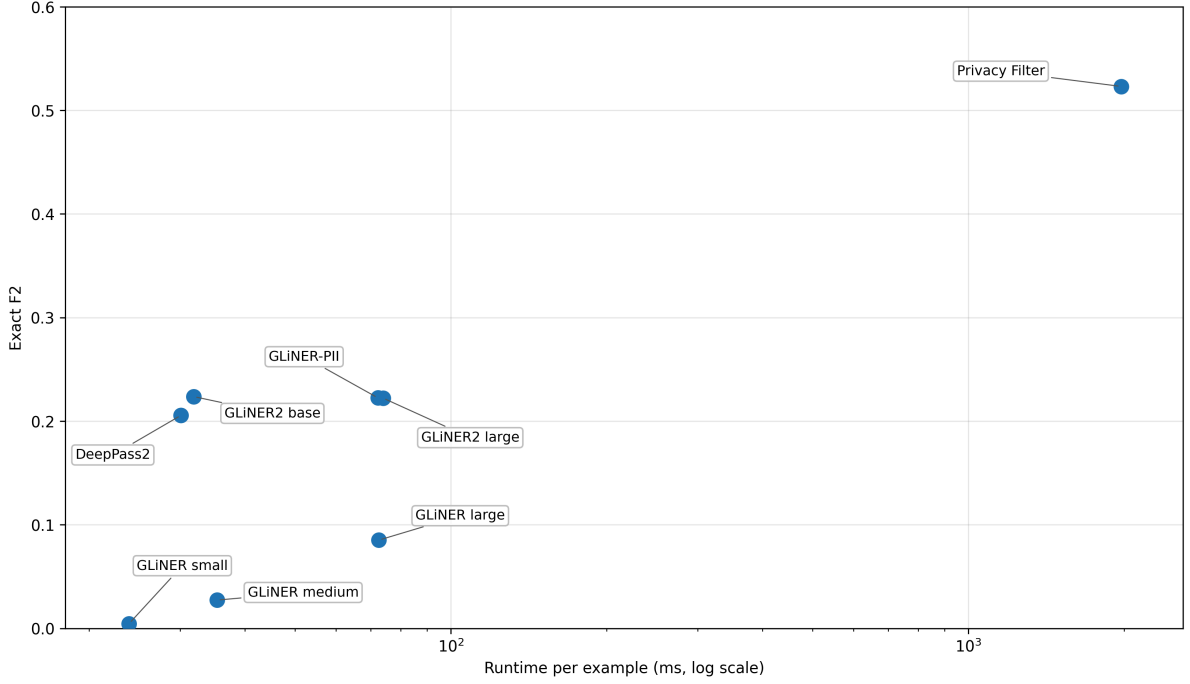


Figure 22: Exact matching F2 versus runtime of zero-shot model.

The partial-matching results in Table 26 provide an additional view of whether models at least detect overlapping spans. The Privacy Filter also leads under partial matching, with partial-matching F2 0.6639 and average overlap 97.22%. DeepPass2 remains a strong partial baseline with F2 0.5046, mainly because it predicts many password-like spans and therefore obtains higher recall. However, both models are less suitable as the main trainable span detector for this thesis: Privacy Filter is too slow for the intended local prototype workflow, and DeepPass2 is more specialized towards password-like detections and produces many false positives. Figure 23 shows the partial-matching F2 ranking.

Model	Precision	Recall	F1	F2	Avg. overlap	Runtime
Privacy Filter	0.4125	0.7833	0.5404	0.6639	97.22%	1972.39 ms
DeepPass2	0.3265	0.5844	0.4189	0.5046	60.07%	30.10 ms
GLiNER2 large	0.4383	0.2258	0.2980	0.2500	94.55%	73.94 ms
GLiNER2 base	0.2536	0.2480	0.2508	0.2491	95.85%	31.87 ms
GLiNER-P11	0.4799	0.2120	0.2941	0.2386	99.16%	72.28 ms
GLiNER large	0.1762	0.0920	0.1209	0.1017	88.36%	72.60 ms
GLiNER medium	0.1845	0.0267	0.0466	0.0322	87.03%	35.36 ms
GLiNER small	0.1894	0.0111	0.0209	0.0136	45.47%	23.90 ms

Table 26: Zero-shot model comparison under partial matching.

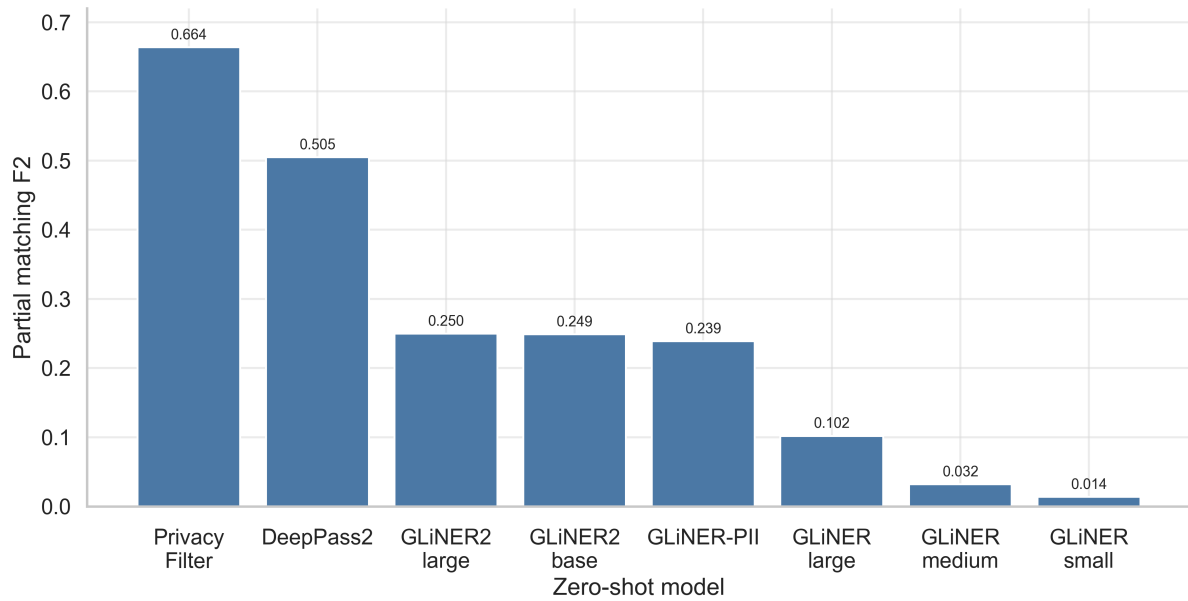


Figure 23: Partial F2 of the zero-shot model comparison.

The comparison changes the interpretation but not the selected trainable model. The Privacy Filter is the strongest zero-shot baseline and shows that a recent externally trained privacy detector can identify many secret-like spans without additional fine-tuning. However, it became available on Hugging Face only after the thesis project had already started and after the GLiNER2-based direction had been defined. It is therefore treated as an additional late baseline rather than as a model that shaped the architecture.

There are also practical reasons not to replace the planned model family with Privacy Filter at this stage. Its runtime is roughly two seconds per example in this benchmark, which is not attractive for an interactive local desktop tool. In addition, the thesis training workflow is built around GLiNER2’s descriptive-label span-detection interface and LoRA fine-tuning on generated span annotations. Privacy Filter is useful as a strong zero-shot comparison point, but it is not the model family selected for controlled adaptation in this thesis.

The decision to fine-tune `fastino/gliner2-base-v1` is therefore based on the combined criteria of runtime, adaptability, project timing and thesis scope, not on zero-shot F2 alone. GLiNER2 base is the strongest GLiNER-family model under exact-matching F2, is much faster than GLiNER2 large, and supports the descriptive label interface used throughout the thesis. It can be fine-tuned directly on the generated span annotations by mapping all secret types to the single model-facing label `Secret`. DeepPass2 and Privacy Filter are useful baselines, but they are less suitable as the main refinement target: DeepPass2 is specialized towards password-like detections, while Privacy Filter is a late, much slower baseline and less aligned with the planned GLiNER2 fine-tuning workflow. Therefore, GLiNER2 base remains the most suitable model for LoRA fine-tuning and local prototype integration.

7.3 GLiNER2 Fine-Tuning Results

The fine-tuned GLiNER2 adapter is evaluated on the synthetic test set. In addition to F1, F2 is reported because recall is weighted more strongly in the final redaction evaluation.

7.3.1 Training Behavior

Figure 24 shows the training and validation loss over the ten training epochs. Both curves decrease substantially during training. The validation loss reaches its lowest value at epoch 7 (0.2392) and then increases slightly in the last two epochs. This suggests that the run benefits from training beyond the first

few epochs, but the final epochs do not improve the validation loss further. The best exported adapter is therefore selected using the trainer’s validation signal rather than by assuming that the final epoch is necessarily optimal.

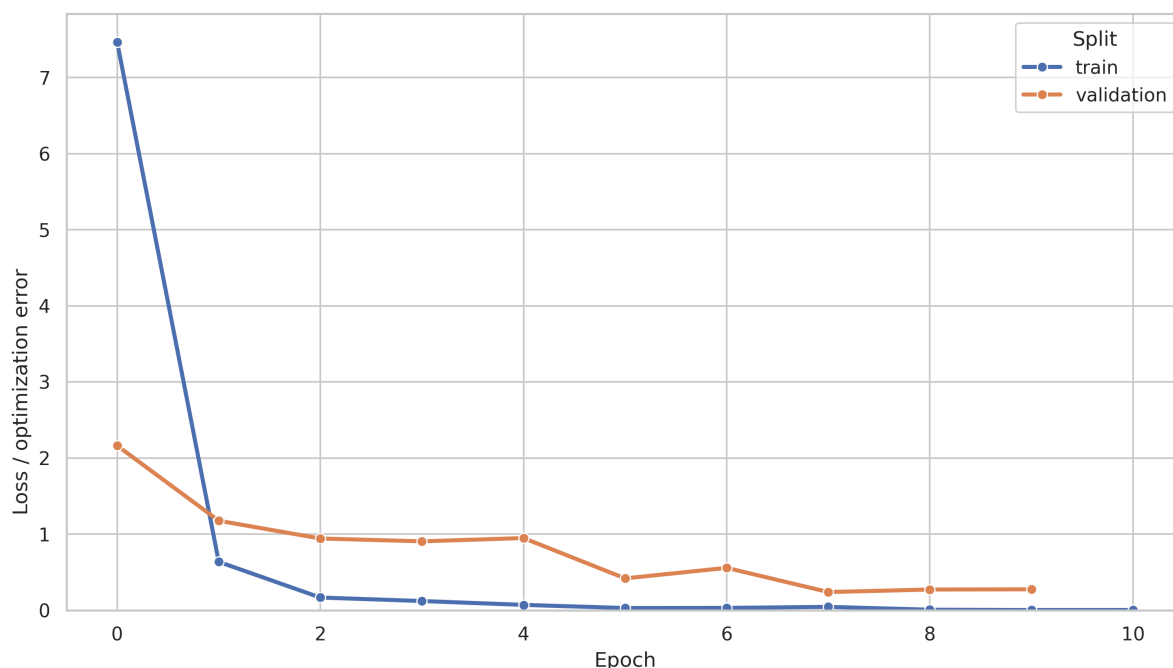


Figure 24: Training and validation loss of the GLiNER2 LoRA fine-tuning run.

7.3.2 Threshold Sensitivity

The evaluation uses a prediction threshold of 0.8. A threshold sweep on the test split shows that the model is not highly sensitive to this threshold over the evaluated range. The exact matching F2 score remains approximately 0.8928 to 0.8929 for thresholds between 0.2 and 0.9, and only changes slightly at 0.95. Because recall remains essentially constant across this range, F2 is also very stable.

Figure 25 shows the class specific score distribution across the candidate spans in the test set. Both the secret and non-secret classes produce a large spike near 1.0 and have very few scores in the middle.

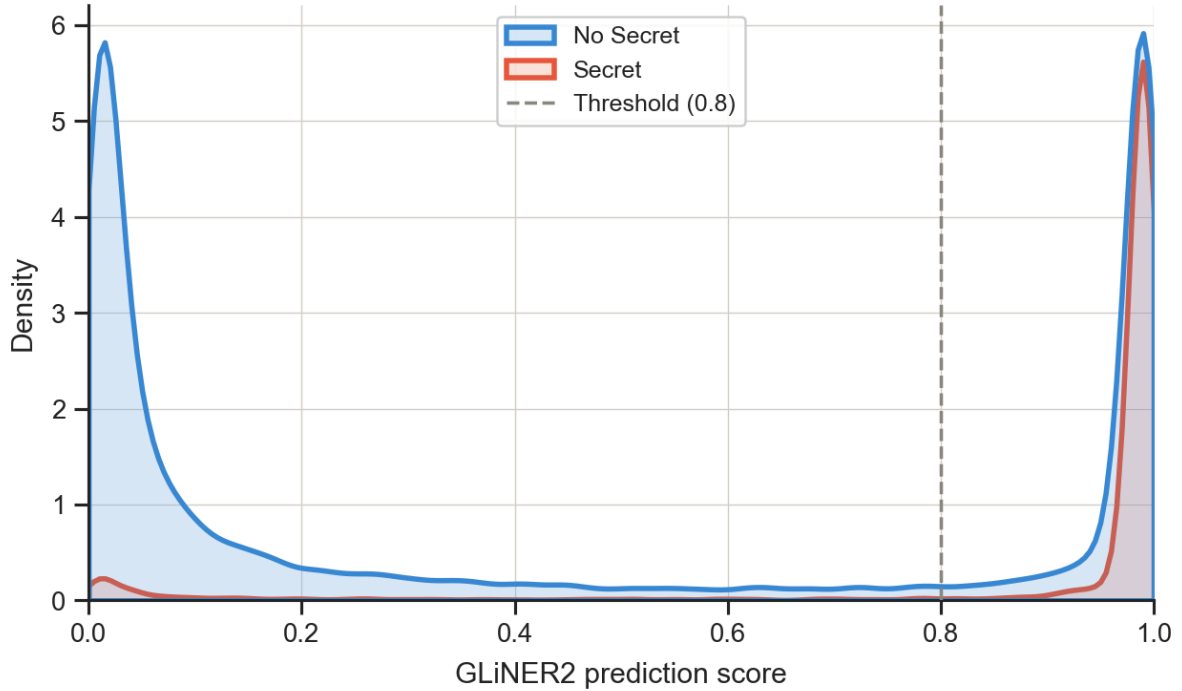


Figure 25: GLiNER2 secret prediction score distribution on the test set.

This stability reflects a model which assigns most predictions a score close to either 0.0 or 1.0, with comparatively few spans receiving medium scores. When predictions are this confident, moving the threshold anywhere in the range 0.2 - 0.9 reclassifies very few spans, so the resulting metrics are nearly identical. Only at 0.95 does the threshold begin to exclude some high-confidence positive predictions, which causes a small drop in F2. This pattern is consistent with a model that has learned to separate the Secret label from all other spans with high confidence on this synthetic test distribution.

7.3.3 Final Synthetic Test Results

Table 27 reports exact and overlap metrics for the fine-tuned model. Exact matching requires the predicted span to match the annotated span boundaries exactly. Overlap matching counts predictions that overlap with the target span and is therefore less strict. The F2 score highlights the recall limitation more strongly than the F1 scores.

Match	Precision	Recall	F1	F2	TP	FP	FN
Exact	0.9978	0.8080	0.8929	0.8400	4,886	11	1,161
Overlap	0.9996	0.8095	0.8946	0.8415	4,895	2	1,152

Table 27: Final GLiNER2 LoRA performance at threshold 0.8.

The model achieves very high exact-matching precision (0.9978) but recall of only 0.8080. The gap is clearer under F2 than under F1: exact-matching F1 is 0.8929, while exact-matching F2 is only 0.8400. The model is therefore precise but too conservative for the redaction use case on its own.

Compared with the regex-only baseline, the fine-tuned model improves exact-matching F1 substantially (0.6156 to 0.8929). However, the F2 value shows that recall remains the main limitation, which motivates the hybrid design: GLiNER2 is a strong high-precision component, but it must be combined with the other candidate-generation stages and a recall-aware final stage.

7.3.4 Best Configuration

The best reported model-based configuration is the GLiNER2 LoRA adapter evaluated at threshold 0.8 and listed in [Table 28](#). The same checkpoint is used in [Section 7.6](#) for the ablation and is the one shipped in the prototype.

Aspect	Value
Base model	fastino/gliner2-base-v1
Adaptation method	LoRA adapter
Label schema	Single label: Secret
Threshold	0.8
Primary metric	F2 for final redaction interpretation F1 for comparison
Test exact-matching precision	0.9978
Test exact-matching recall	0.8080
Test exact-matching F1	0.8929
Test exact-matching F2	0.8400

Table 28: Best reported GLiNER2 configuration and its exact value-level metrics.

7.4 Candidate Classifier Results

Secrets-Sentinel is used as an optional upstream candidate-level signal rather than as the final decision mechanism. In the final meta-classifier experiments, its contribution is represented through the `classification_score` feature in the `with_classifier_feature` variant. A separate standalone benchmark of Secrets-Sentinel against alternative classifier backends was not performed in the final project setup.

The relevant empirical question is therefore whether including the classifier score improves the final LightGBM decision stage. This comparison is reported in [Section 7.5](#). The result is important for deployment because the classifier-aware variant requires running an additional upstream model, whereas the classifier-free variant can make the final decision using only regex, entropy, GLiNER2, candidate shape and context features.

7.5 Meta-Classifier Results

The meta-classifier experiment compares two LightGBM variants: one with the optional Secrets-Sentinel `classification_score` feature and one without it. Both variants use the reduced fixed feature schema described in [Section 6.14](#) and are evaluated with a grouped train/validation/test set by `case_id`.

7.5.1 Classifier Performance

[Table 29](#) reports the candidate-level results for both variants.

Variant	Train F1	Val F1	Test precision	Test recall	Test F1	Avg. Precision	Train-Val F1 gap	Val-Test F1 gap
with classifier feature	0.9966	0.9860	0.9747	0.9917	0.9831	0.9983	0.0106	0.0029
without classifier feature	0.9955	0.9849	0.9765	0.9926	0.9845	0.9984	0.0106	0.0004

Table 29: Final LightGBM candidate-level meta-classifier results.

As shown in Table 29 both variants perform strongly on the candidate-level task. The classifier-aware variant reaches a test F1 of 0.9831 and test average precision of 0.9983. The classifier-free variant reaches a slightly higher test F1 of 0.9845 and test average precision of 0.9984. The difference is small, but it is consistent with the operational preference for the classifier-free variant: it performs marginally better on the grouped candidate test set and avoids the runtime and integration dependency of running Secrets-Sentinel upstream.

The train-validation F1 gap is 0.0106 for both variants. The validation-test gap is small in both cases and is especially small for the classifier-free variant (0.0004). This suggests mild but acceptable overfitting: the models fit the training candidates slightly better than the validation candidates, but performance does not collapse on the grouped test set. The confusion matrix, shown in Figure 26, displays the result of the selected classifier-free LightGBM meta-classifier on the test set.

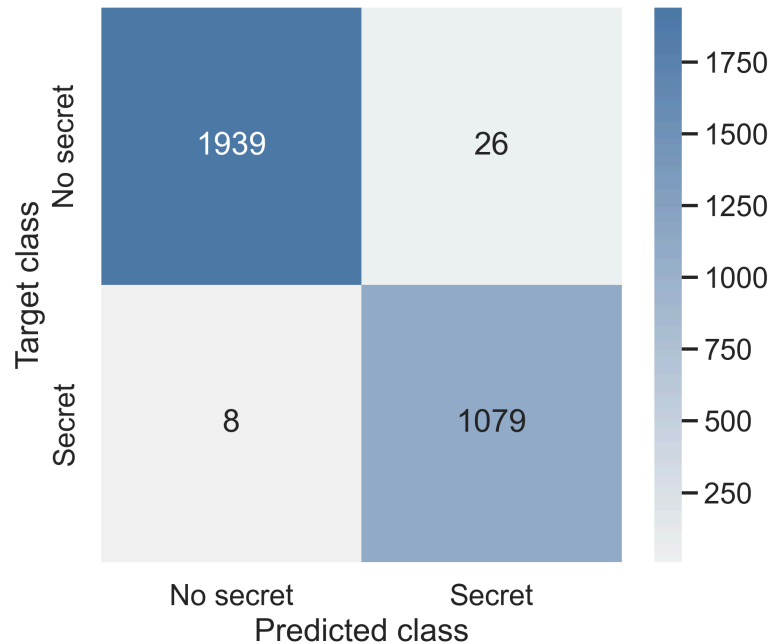


Figure 26: Confusion matrix of the selected LightGBM meta-classifier.

The selected classifier-free variant produces 1079 true positives, 26 false positives, 8 false negatives and 1939 true negatives on the test set. The very small number of false negatives is important for the redaction task because a false negative leaves a candidate unredacted. At the same time, the number of false positives remains low, which means the final decision layer does not simply accept all candidates generated by the high-recall upstream stages.

The most important features of the classifier-free variant are candidate-shape and confidence signals. The strongest transformed features shown in Figure 27 include `candidate_special_ratio`,

candidate_length, candidate_digit_ratio, candidate_entropy, best_gliner_entity_score and candidate_score. The features are ordered by the importance during the classifier evaluation. This supports the intended design of the meta-classifier: it does not rely on raw candidate text, but combines structural evidence, upstream model confidence, entropy and stage provenance.

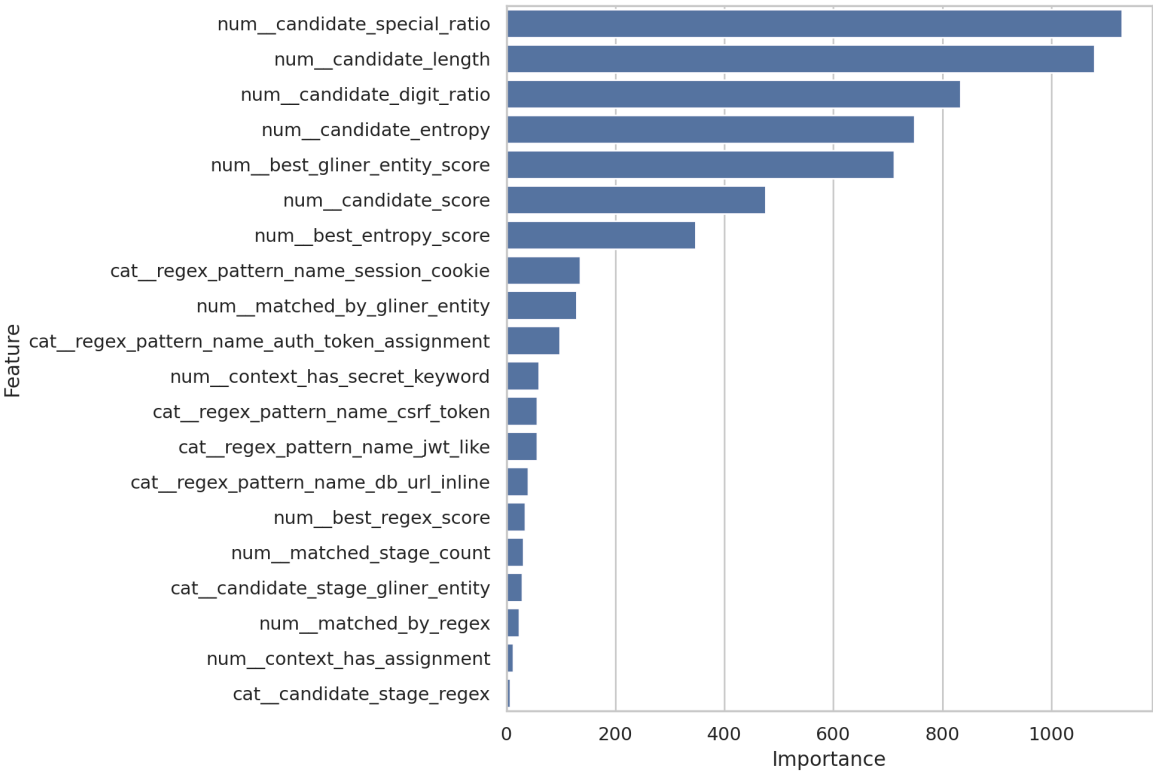


Figure 27: Best transformed feature importances of selected LightGBM meta-classifier.

The learning curves in Figure 28 show steadily decreasing binary log loss and high average precision on both train and validation data. The validation curves remain close to the training curves, supporting the interpretation that the feature set is stable on the grouped candidate split.

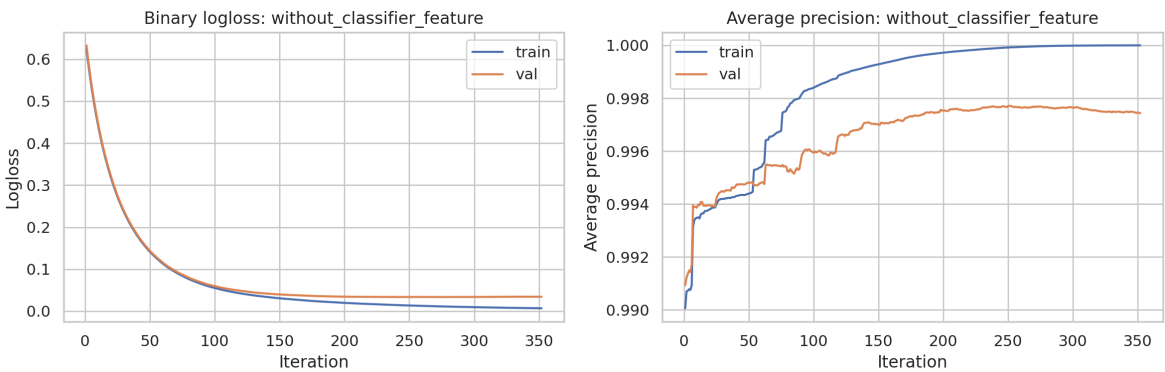


Figure 28: LightGBM learning curves of the selected classifier-free meta-classifier.

The classifier-aware variant remains useful as an optional artifact for a pipeline configuration that already computes Secrets-Sentinel scores. However, the final default candidate is the without_classifier_feature variant because it is simpler to deploy and slightly stronger on the evaluated test set.

7.5.2 Threshold Selection

The initial meta-classifier experiments used the default decision threshold of 0.5. This is a common default for binary classifiers, but it is not necessarily the optimum for all use-cases. In this use case, a false negative is more critical than a false positive because an undetected secret remains visible in the output. Therefore, the final decision threshold should be selected explicitly rather than accepted as a fixed default.

To analyze the precision-recall trade-off, the LightGBM meta-classifier was evaluated across multiple thresholds and labeling schemes shown in Figure 29. The exact scheme counts a candidate as a true positive only when its span matches the target secret exactly. The overlap-aware schemes relax this criterion, treating a candidate as positive when it covers a sufficient share of the target secret span. This better reflects the redaction objective, since a slightly wider span will still remove the sensitive value in full.

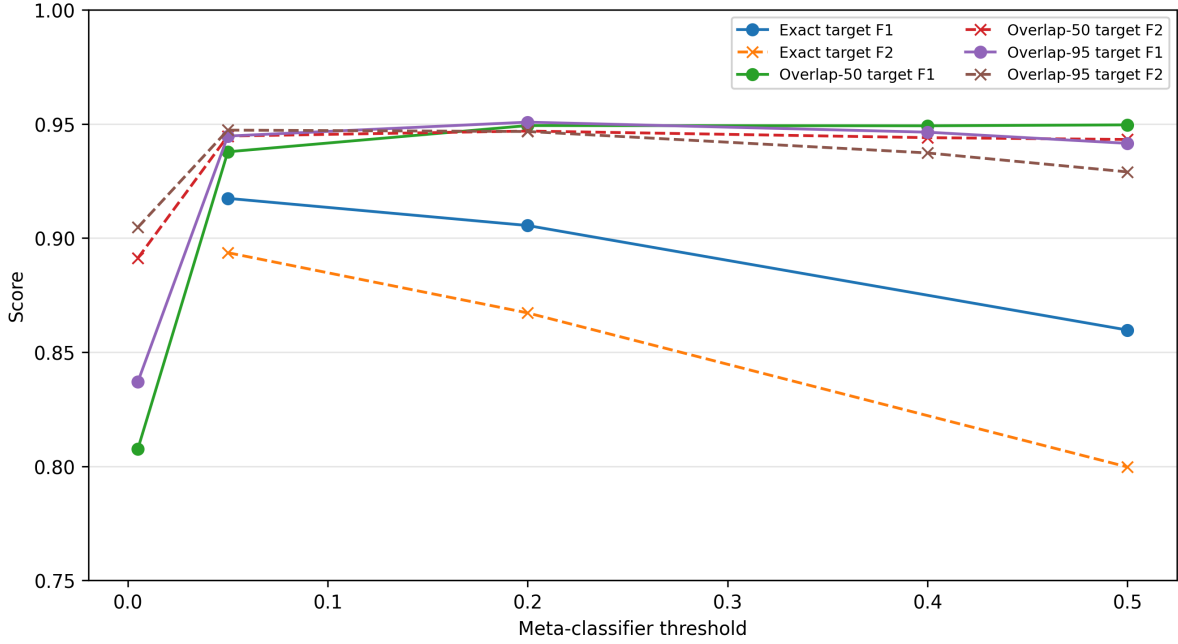


Figure 29: Meta-classifier threshold sweep on the synthetic test set using exact matching.

The original exact-target classifier remains much weaker than the overlap-aware variants. At threshold 0.05, the exact-target model reaches exact-matching F1 0.9174 and F2 0.8936. In contrast, the overlap-aware variants reach exact-matching F1 values around 0.95 and substantially higher F2 values. This indicates that the target definition, not only the threshold, has a substantial effect on the final operating point.

Training target	Criterion	Threshold	Precision	Recall	F1	F2	Evaluation
Exact target	Best F1/F2	0.05	0.9601	0.8785	0.9174	0.8936	Exact
Overlap-50 target	Best F1	0.50	0.9604	0.9390	0.9496	0.9432	Exact
Overlap-50 target	Best F2	0.20	0.9535	0.9453	0.9493	0.9469	Exact
Overlap-95 target	Best F1	0.20	0.9577	0.9439	0.9508	0.9467	Exact
Overlap-95 target	Best F2	0.05	0.9404	0.9491	0.9447	0.9473	Exact

Table 30: Best criterion of the LightGBM classifier on the synthetic test set.

Table 30 reports the best criterion for each training target. The exact results show a clear trade-off: higher thresholds preserve precision, while lower thresholds recover recall. The overlap-50 target reaches its best exact-matching F1 at threshold 0.5, with precision 0.9604, recall 0.9390, F1 0.9496 and F2 0.9432. However, its best F2 occurs at threshold 0.2, where recall increases to 0.9453 while precision remains

high at 0.9535. The overlap-95 target reaches the highest exact-matching F1 in this sweep: F1 0.9508 at threshold 0.2. Its best F2 is reached at threshold 0.05, where recall increases to 0.9491 with precision 0.9404.

The 90% overlap evaluation in Figure 30 confirms the same pattern under a redaction-oriented 90% overlap criterion. The overlap-aware target variants remain clearly stronger than the original exact target, and lowering the threshold improves recall while precision decreases gradually.

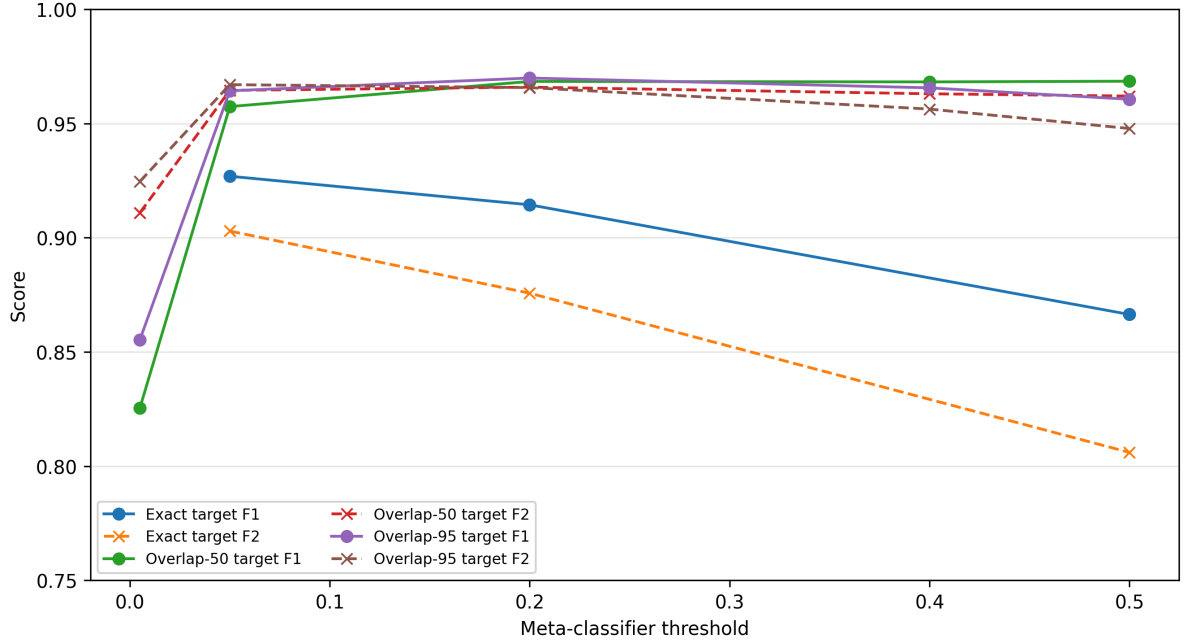


Figure 30: Meta-classifier threshold sweep on the synthetic test set using partial matching.

Table 31 reports the corresponding 90%-overlap operating points. These results are important because they show that the improvement is not only caused by weak partial matches. Even when a prediction must cover at least 90% of the target span, the overlap-aware models reach F1 values around 0.97 and F2 values above 0.96. The overlap-95 target is especially relevant because it remains close to exact-span behavior during training, but still rewards candidates that nearly cover the full sensitive value.

Training target	Criterion	Threshold	Precision	Recall	F1	F2	Evaluation
Exact target	Best F1/F2	0.05	0.9700	0.8875	0.9269	0.9029	90% overlap
Overlap-50 target	Best F1	0.50	0.9795	0.9577	0.9685	0.9620	90% overlap
Overlap-50 target	Best F2	0.20	0.9726	0.9643	0.9684	0.9659	90% overlap
Overlap-95 target	Best F1	0.20	0.9770	0.9630	0.9699	0.9657	90% overlap
Overlap-95 target	Best F2	0.05	0.9599	0.9687	0.9643	0.9670	90% overlap

Table 31: Best 90%-overlap criterion of the LightGBM classifier.

These results support two decisions for the final pipeline. First, the meta-classifier should be trained with an overlap-aware target rather than only with strict exact labels. Second, the deployment threshold should be treated as an explicit operating parameter. A higher threshold is preferable when precision and minimal over-redaction are the main goal. A lower threshold is preferable when recall is more important.

7.6 Pipeline Results and Ablation

Figure 31 summarises the exact-matching F2 progression across the five evaluated configurations. Each bar represents the best result achieved by that configuration on the synthetic test set. The chart makes the cumulative effect of adding components visible at a glance: each stage of the hybrid design improves on the previous one, and the final pipeline with the LightGBM meta-classifier achieves the highest F2 of 0.949.

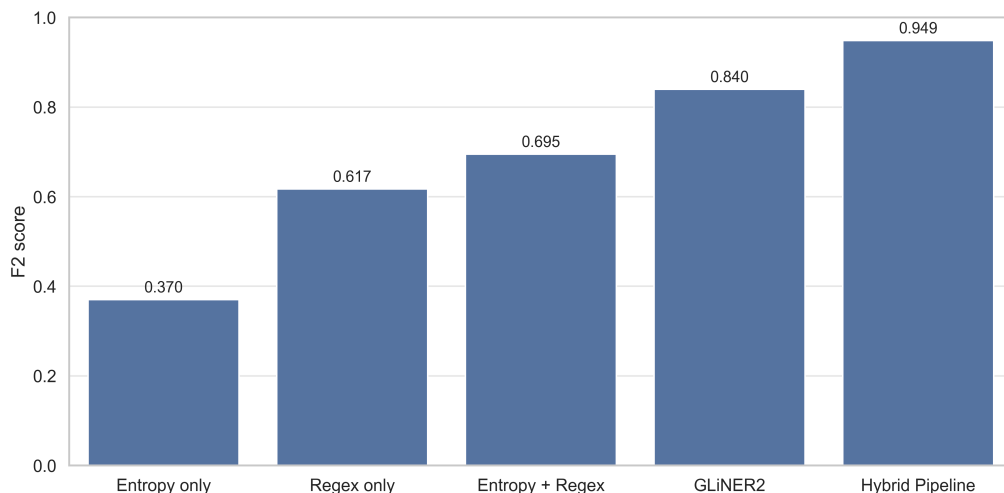


Figure 31: Exact-matching F2 progression across pipeline configurations.

Entropy alone is the weakest configuration (F2 0.370), limited by its high false-positive rate. Regex alone reaches F2 0.617 by catching structured known-format credentials precisely. Combining regex and entropy raises recall but keeps precision low, resulting in F2 0.695. The fine-tuned GLiNER2 adapter alone reaches F2 0.840 by adding context-sensitive detection. The full hybrid pipeline (regex, entropy and GLiNER2 candidates filtered by the LightGBM meta-classifier) achieves F2 0.949, confirming that the combination of complementary signals with a learned final decision layer is the decisive factor.

The final hybrid-pipeline benchmark compares four deployment-relevant configurations on the synthetic test set. All runs share the same upstream candidate generation stages, so the comparison also serves as a late-stage ablation that isolates the effect of adding Secrets-Sentinel and the LightGBM meta-classifier. Exact matching is the strict primary view and F2 is the main aggregate metric, because a false negative leaves a secret visible. The meta-classifier variants use the overlap-aware LightGBM classifier with a threshold of 0.1. Table 32 and Table 33 report exact- and partial-matching results, and Figure 32 visualises the evaluated metrics (precision, recall, F1 and F2) per run.

Run	Precision	Recall	F1	F2	TP	FP	FN	Ø Runtime
No classifier	0.2125	0.9649	0.3483	0.5649	5835	21626	212	60 ms
Secrets-Sentinel	0.2934	0.8851	0.4407	0.6307	5352	12892	695	216 ms
Meta + Sentinel	0.9506	0.9487	0.9497	0.9491	5737	298	310	261 ms
Meta only	0.9498	0.9486	0.9492	0.9488	5736	303	311	113 ms

Table 32: Exact-matching performance of the final hybrid pipeline configurations.

Run	Precision	Recall	F1	F2	TP	FP	FN
No classifier	0.2198	0.9982	0.3603	0.5843	6036	21425	11
Secrets-Sentinel	0.3041	0.9175	0.4568	0.6538	5548	12696	499
Meta + Sentinel	0.9758	0.9739	0.9748	0.9743	5889	146	158
Meta only	0.9755	0.9742	0.9748	0.9745	5891	148	156

Table 33: Partial-matching performance of the final hybrid pipeline configurations.

Without a final classifier layer, the combined upstream candidate stages are highly recall-oriented but produce many false positives: the `No classifier` run reaches exact-matching recall 0.9649 at only 0.2125 precision, with 21626 false positives. Adding `Secrets-Sentinel` alone reduces false positives to 12892, but exact-matching F2 remains low at 0.6307 and recall drops to 0.8851, confirming that a standalone candidate classifier is not sufficient as the final decision mechanism.

The `LightGBM` meta-classifier is the dominant quality improvement. Compared with `Secrets-Sentinel` alone, the `Meta + Sentinel` run reduces false positives from 12892 to 298 and raises exact-matching F2 from 0.6307 to 0.9491, with recall recovering to 0.9487. The overlap-aware training target and threshold of 0.1 move the operating point toward higher recall while retaining high precision, which aligns the decision layer with the redaction objective.

The two meta variants are nearly identical in quality. `Meta + Sentinel` leads by a marginal exact-matching F2 (0.9491 versus 0.9488), while `Meta only` is slightly stronger under partial matching (0.9745 versus 0.9743). Because the differences are negligible, runtime and deployment complexity become decisive: `Meta only` avoids the `Secrets-Sentinel` dependency and is more than twice as fast (113 ms versus 261 ms). It is therefore selected as the preferred final configuration.

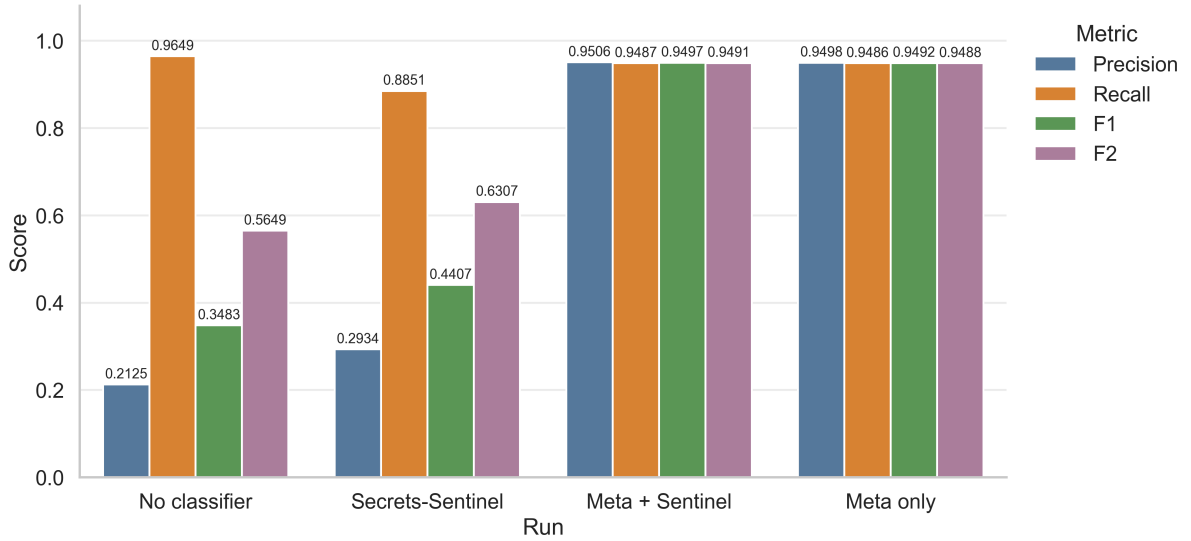


Figure 32: Exact-matching precision, recall, F1 & F2 of the hybrid pipeline configurations.

7.7 Stage-Wise Contribution Analysis

The final hybrid pipeline combines regex, entropy and GLiNER2 candidate generation before the `LightGBM` meta-classifier makes the final keep/drop decision. To better understand the contribution of each candidate stage, the stage-wise ablation distinguishes between three measures. *Independent coverage* is the share of target secrets a stage detects before the final classifier, regardless of what the other stages detect. *Unique coverage* is the share of target secrets detected exclusively by that stage and no other. *Final true-positive share* is the share of accepted final detections whose retained candidate is attributed to that stage after merging. Table 34 and Figure 33 report these three measures for each stage.

Stage	Independent coverage	Unique coverage	Final TP share
Regex	70.23%	18.31%	66.14%
GLiNER2	85.84%	15.99%	33.39%
Entropy	46.65%	0.68%	0.47%

Table 34: Stage-wise exact-matching contribution of the individual candidate stages.

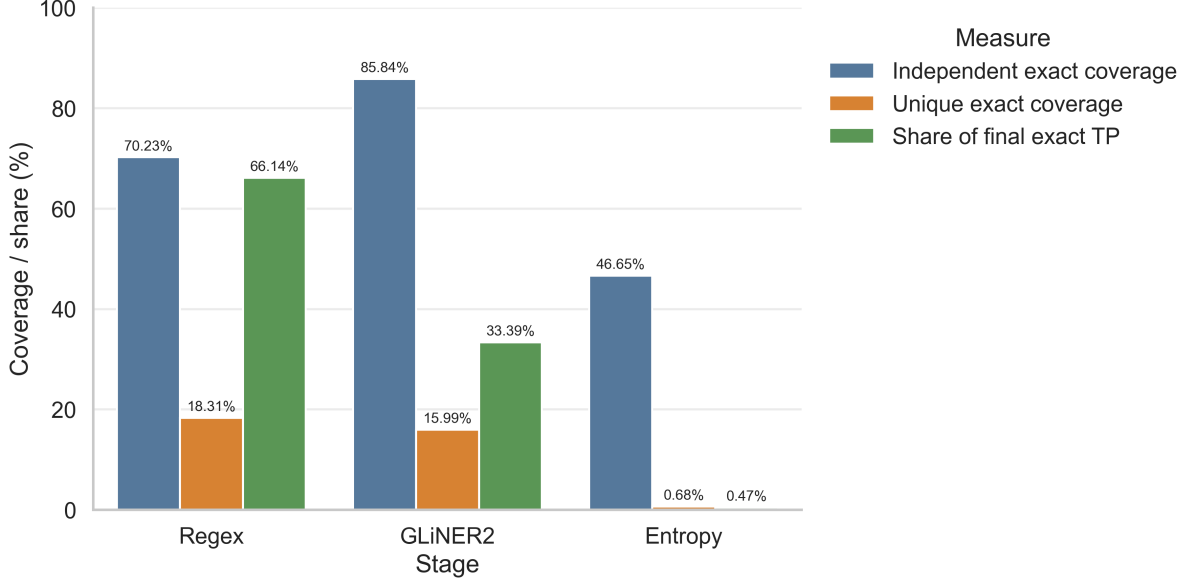


Figure 33: Stage-wise exact-matching contribution of each candidate stage.

The pipeline does not rely on a single dominant detector. GLiNER2 has the highest independent coverage (85.84%), meaning it reaches the most target secrets before classification. Regex has the largest final true-positive share (66.14%), which is expected because overlapping candidates are merged and the retained span is frequently attributed to the regex match even when GLiNER2 also detected the same value. The unique-coverage values confirm that both stages are necessary: regex uniquely covers 18.31% of target secrets and GLiNER2 uniquely covers 15.99%, so removing either would leave a meaningful share of secrets undetected. Entropy contributes negligible unique coverage (0.68%) and a final true-positive share of only 0.47%, but remains useful as a broad recall-oriented signal and as an input feature for the meta-classifier.

This supports the final architecture: regex and GLiNER2 provide complementary detection with comparable unique contributions, while entropy is better interpreted as a noisy candidate-generation and feature signal than as a standalone detector.

7.8 Error Analysis

For the selected final Meta only configuration, the evaluation on test set reports 5736 exact matching true positives, 303 false positives and 311 false negatives out of 6047 target secret spans.

7.8.1 Thresholded Partial Matching

Broad partial matching counts any sufficiently overlapping prediction as a success. To make this more informative for redaction, the final benchmark also reports overlap-threshold metrics. These metrics require that a prediction covers at least a specified share of the target span before it counts as a true positive. For example, the $\geq 90\%$ criterion accepts only predictions that cover at least 90% of the target secret value.

Criterion	Precision	Recall	F1	F2	TP	FP	FN
Exact	0.9498	0.9486	0.9492	0.9488	5736	303	311
Any overlap	0.9755	0.9742	0.9748	0.9745	5891	148	156
$\geq 50\%$ target coverage	0.9737	0.9724	0.9730	0.9726	5880	159	167
$\geq 70\%$ target coverage	0.9704	0.9691	0.9697	0.9693	5860	179	187
$\geq 80\%$ target coverage	0.9700	0.9687	0.9694	0.9690	5858	181	189
$\geq 90\%$ target coverage	0.9692	0.9679	0.9686	0.9682	5853	186	194
$\geq 95\%$ target coverage	0.9685	0.9673	0.9679	0.9675	5849	190	198

Table 35: Exact, partial and thresholded-overlap metrics of selected final pipeline.

Figure 34 visualises how F1 and F2 change as the coverage criterion tightens from any overlap to exact matching. The two lines follow a shallow, nearly flat decline across the partial-matching range and then drop more noticeably only at the exact-matching step. Between any overlap and the strictest partial criterion ($\geq 95\%$), F1 falls by only 0.0069 (from 0.9748 to 0.9679) and F2 by 0.0070. The jump to exact matching then accounts for the remaining drop to 0.9492 and 0.9488 respectively. This pattern confirms that most non-exact but overlapping successes cover a large share of the target secret rather than merely touching it at the boundary, and that the gap between partial and exact matching is driven by span-boundary imprecision rather than by weak or marginal overlaps.

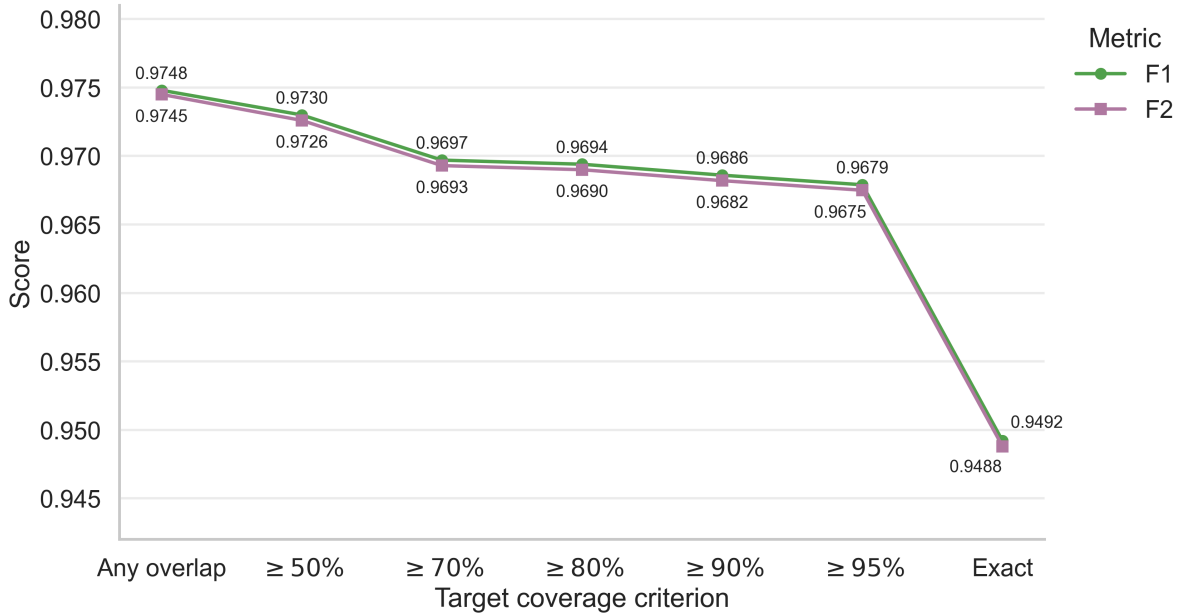


Figure 34: F1 and F2 across overlap coverage thresholds of final pipeline.

7.8.2 Per-Category Performance

The final hybrid benchmark is evaluated in secret-only mode. The pipeline therefore predicts the generic redaction label Secret rather than a fine-grained category such as Password, APIKey or DatabaseURL. However, the generated samples retain the original secret type of each target span as metadata. This makes it possible to analyse type-level behaviour of the final pipeline without treating the system as a fine-grained secret classifier.

The results in this section are based on the final selected pipeline configuration: regex, entropy and the fine-tuned GLiNER2 adapter are enabled, and the LightGBM meta-classifier is used as the final decision layer. The per-category breakdown in this section should be read as an analysis of the final test result by actual secret type, not as a category-prediction task.

Most actual secret types are detected with high exact matching F2. Provider-style tokens and structured keys such as APIKey, AWSAccessKey, and GitHubToken reach very high exact matching F2 scores. This indicates that the final hybrid pipeline handles many structured token formats reliably on the synthetic test split.

The weakest exact matching secret type is RedisURL. Out of 86 target spans, only 24 are true positives with 5 false positives and 62 are false negatives, resulting in an F2 score of 0.3217. Other comparatively weak categories include PrivateKey, KafkaSASLPassword, SMTPURL, DatabaseURL, AMQPConnectionString and BasicAuthURL. These types are mostly long, compounded values with separators and embedded credentials. Exact matching is particularly strict for such values because missing a prefix, suffix or embedded component turns the result into a false negative even if the most sensitive substring was masked.

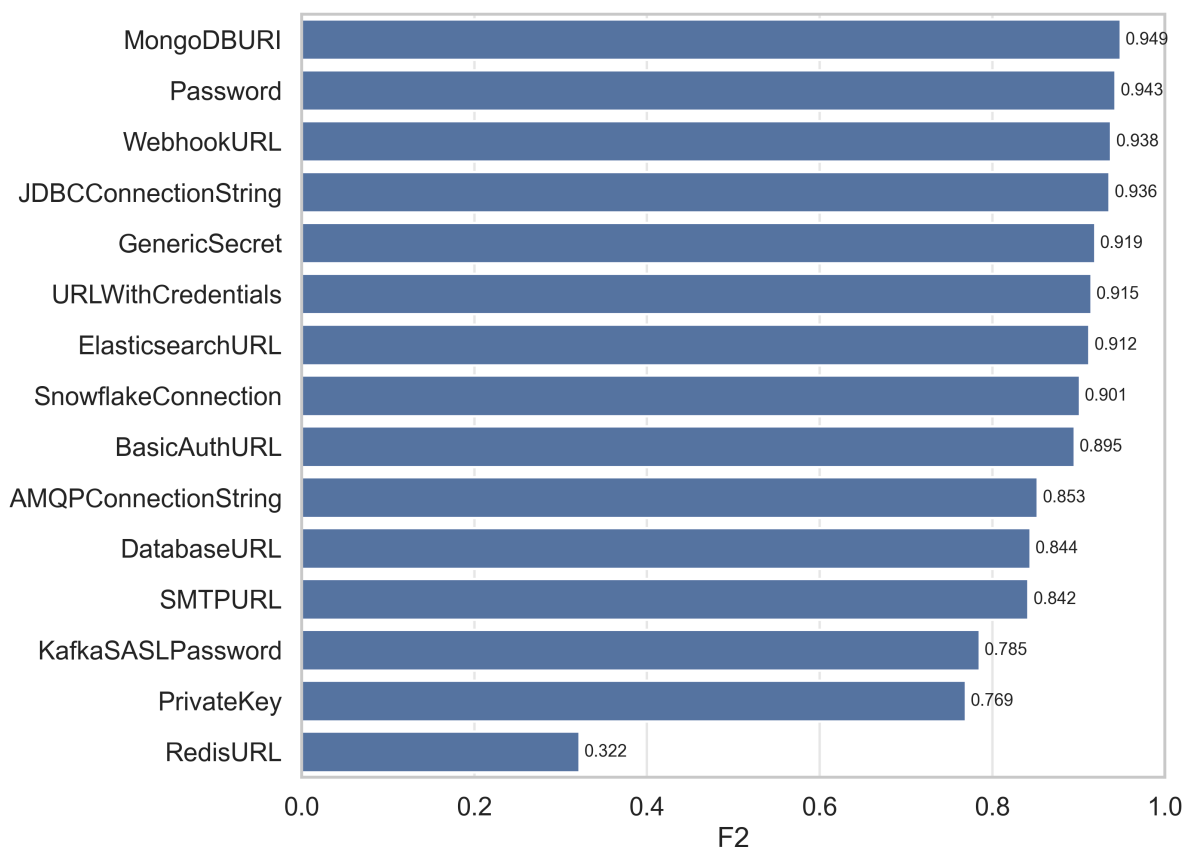


Figure 35: Actual secret types with the lowest exact matching F2 scores.

Secret type	Count	TP	FP	FN	Recall	F2
RedisURL	86	24	5	62	0.2791	0.3217
PrivateKey	51	38	5	13	0.7451	0.7692
KafkaSASLPassword	101	79	20	22	0.7822	0.7853
SMTPURL	99	83	14	16	0.8384	0.8418
DatabaseURL	77	64	7	13	0.8312	0.8443
AMQPConnectionString	103	88	16	15	0.8544	0.8527
BasicAuthURL	100	89	8	11	0.8900	0.8954
SnowflakeConnection	108	95	0	13	0.8796	0.9013

Table 36: Lowest exact matching F2 secret types with at least 20 target spans.

7.8.3 Error Pattern Analysis

The enriched error cases from the evaluation make it possible to inspect the properties of remaining false positives and false negatives. Table 37 summarises the main statistics and Figure 36 shows the full entropy distributions for both error types.

Error	Count	Avg. length	Median length	Avg. entropy	Regex support	Entropy support
False negatives	311	68.2	50.0	4.53	77.8%	88.7%
False positives	303	59.4	64.0	4.78	55.4%	96.7%

Table 37: Aggregate properties of exact-matching errors.

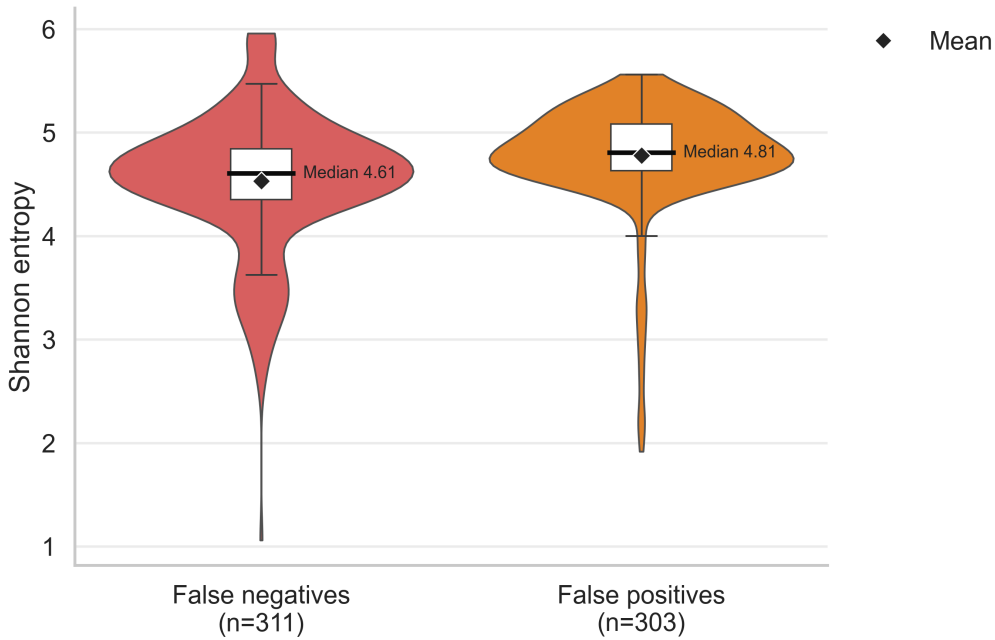


Figure 36: Shannon entropy distribution of false negatives and false positives.

False positives exhibit slightly higher entropy than false negatives, with medians of 4.81 and 4.61 respectively. However, as Figure 36 shows, the two distributions overlap substantially across the full interquartile range. A random false positive has higher entropy than a random false negative in only 66.3% of cases. Entropy alone therefore cannot reliably separate the two error types.

This is most evident for missed passwords, whose average entropy is only 3.29. Lowering an entropy threshold would not recover these cases and would instead increase false positives further. They require context-sensitive detection rather than a signal based on character randomness. Missed RedisURL instances (62 cases) present a different challenge: their entropy is not low, but the full credential-bearing URL span is boundary-sensitive and the classifier rejects the candidate when the supporting signals are not sufficiently strong.

False positives are mostly high-entropy, secret-like hard negatives. Entropy support is present for 96.7% of false positives, which confirms the role of entropy as a useful but noisy candidate generator: it contributes recall-oriented candidates that the final decision layer must then filter. The per-type breakdown in Table 38 confirms these two failure modes in more detail.

Missed secret type	Count	Avg. length	Avg. entropy
RedisURL	62	45.0	4.42
Password	44	13.1	3.29
KafkaSASLPassword	22	46.0	4.84
GenericSecret	19	57.7	5.07
SMTPURL	16	77.8	4.65
AMQPConnectionString	15	70.2	4.68

Table 38: Most frequent missed secret types.

The missed-type distribution is dominated by credential-bearing URLs (RedisURL, SMTPURL, AMQPConnectionString) and low-entropy passwords. This points to two distinct failure modes: long compound URL values whose exact span boundaries are difficult, and low-entropy passwords that require context rather than entropy. Both are concrete targets for future data generation and rule design rather than evidence of a fundamental limitation of the architecture.

7.8.4 Qualitative Error Examples

The aggregate error statistics show which types of errors remain, but individual examples help explain why these cases are difficult. The error examples in Table 39 are shortened and do not contain functional credentials.

Error type	Shortened example	Explanation
False Negative: credential-bearing URL	redis://:hunter 242@db.internal:6379/10	Looks like an internal service URL with a low-entropy password. Regex and entropy may generate a candidate, but the classifier rejects it when the combined evidence is not strong enough.
False Negative: private-key block	-----BEGIN PGP PRIVATE KEY BLOCK----- [...]	Only an internal high-entropy substring is predicted while the full annotated span is missed under exact matching. The evaluation expects the complete block, so partial detection does not count.
False Positive: dummy connection string	redis://:dummy-pass- [...] @localhost:6379/0	Structurally identical to a real credential-bearing URL and supported by both regex and entropy. The terms dummy and localhost signal a non-functional value, but the distinction is subtle.
False Positive: high-entropy config value	nSIxi8bV...R4HXY=[...]	High entropy and secret-like character composition trigger candidate generation. This illustrates why entropy is a useful recall signal but too noisy to act as a standalone decision criterion.

Table 39: Representative shortened error examples from the selected final pipeline.

These examples support the quantitative findings. Remaining false negatives are not only caused by missing obvious tokens, they include low-entropy credential-bearing URLs and long structured secrets whose exact span boundaries are difficult. Remaining false positives are mostly secret-like hard negatives that intentionally resemble real credentials. This confirms the role of the final classifier: it must balance

two contradictory risks, rejecting too many true secrets and over-redacting realistic but non-sensitive technical values.

7.8.5 Effect of Entropy Threshold

The entropy-threshold sweep quantifies the precision-recall trade-off introduced by the entropy candidate stage in isolation and in combination with regex. Four thresholds were evaluated: 3.0, 3.5, 4.0 and 4.5. The minimum candidate length was fixed at 16 characters and the regex catalogue was unchanged across all runs. All metrics in Figure 37 refer to exact matching and the dashed line marks the regex-only baseline F2 of 0.617.

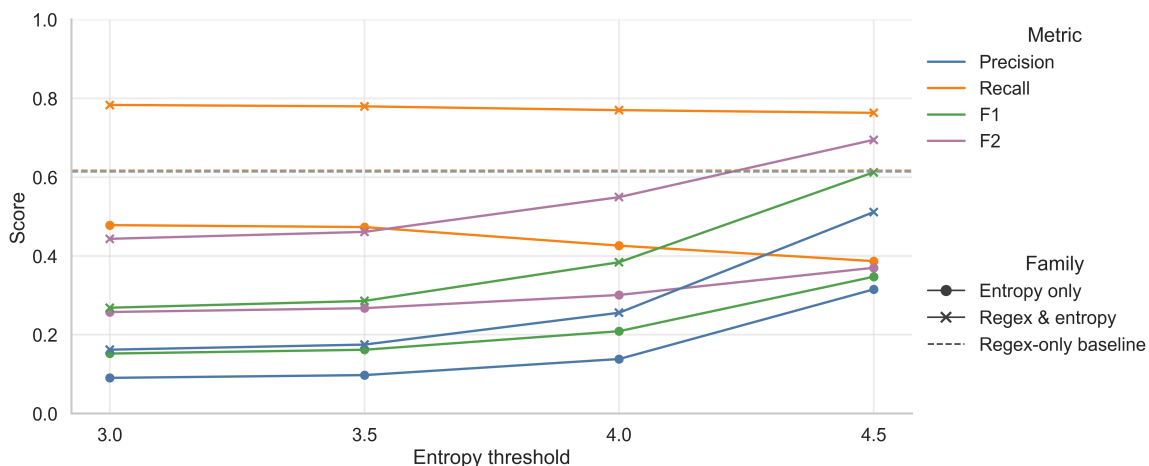


Figure 37: Exact-matching metrics across entropy thresholds.

Figure 37 shows two families. *Entropy only* treats entropy as the sole detection signal. Even at the most permissive threshold (3.0), it achieves only a precision of 0.091 and an exact-matching F2 of 0.258, despite recall of 0.478. Tightening the threshold to 4.5 raises precision to 0.315 but reduces recall to 0.387 and F2 to 0.370. Entropy alone therefore stays well below the regex-only baseline (F2 0.617) at every threshold.

Regex & entropy combines the two candidate generators. At threshold 3.0, recall rises to 0.783 but precision collapses to 0.162, yielding an exact-matching F2 of 0.443. Raising the threshold to 4.5 reduces the false positive pressure: precision recovers to 0.511, recall drops only slightly to 0.763, and F2 reaches 0.695, approaching but not reaching the regex-only baseline. Neither family crosses the regex-only F2 baseline without a final classification layer, the combined variant at 4.5 comes closest.

The final hybrid benchmark uses an entropy threshold of 3.5 for upstream candidate generation. At this setting the combined variant has recall of 0.780 and F2 of 0.461, which is deliberately recall-oriented: the threshold controls which random-looking values are offered to later stages, not the final redaction decision. This is why a lower threshold is still useful inside the full hybrid pipeline even though regex plus entropy alone has poor precision at that setting, the LightGBM meta-classifier filters the resulting false positives in the final step.

7.9 Runtime Observations

Runtime was measured on the synthetic test set alongside the quality metrics. The values in Table 40 reflect pipeline-comparison timings recorded in the Kaggle compute environment and should not be interpreted as final desktop-prototype latency on user hardware.

Run	Avg. runtime	Exact-matching F2	Comment
No classifier	60 ms	0.5649	Fastest, but produces too many false positives.
Secrets-Sentinel	216 ms	0.6307	Improves precision but adds substantial runtime.
Meta + Sentinel	261 ms	0.9491	High quality but slowest due to the Secrets-Sentinel dependency.
Meta only	113 ms	0.9488	Best quality-to-runtime trade-off.

Table 40: Average runtime and performance of the hybrid pipeline configurations.

The selected `Meta only` configuration offers the best quality-to-runtime trade-off. It is approximately 2.3 times faster than `Meta + Sentinel` while achieving nearly identical exact-matching F2 (0.9488 versus 0.9491), which reinforces the decision to avoid the Secrets-Sentinel dependency in the default configuration. A runtime benchmark on the packaged desktop prototype is left to future work ([Section 9.2](#)).

8 Discussion

This chapter places the results in the broader context of the literature, discusses threats to validity, and reflects on the engineering and research trade-offs that arose during the project. The chapter intentionally separates the interpretation of the results from the results themselves so that readers who only want the numbers can stop after [Section 7](#).

8.1 Interpretation of the Results

The strongest practical configuration is the classifier-free `Meta` only hybrid pipeline, combining regex, entropy and GLiNER2 candidate generation with an overlap-aware LightGBM meta-classifier at threshold 0.1. On the synthetic test set it reaches exact-matching precision 0.9498, recall 0.9486, F1 0.9492 and F2 0.9488, under partial matching it reaches F2 0.9745. The full breakdown is reported in [Table 32](#) and [Table 33](#).

The key finding is that the improvement does not come from combining more detectors, but from separating broad candidate generation from a learned keep/drop decision. The unfiltered union of regex, entropy and GLiNER2 produces very high recall, but the number of false positives is too large for a usable redaction workflow: the `No classifier` run reaches exact-matching recall 0.9649 at precision only 0.2125, with 21626 false positives. The LightGBM meta-classifier reduces these to 303 while keeping recall at 0.9486. That is an improvement that neither adding more rules nor adjusting thresholds alone could achieve.

The first version of the decision layer solved the false-positive problem but was too conservative at the default threshold. The threshold sweep and retraining with overlap-aware targets, reported in [Section 7.5.2](#), show that this was not a fundamental limitation of the architecture. Once trained and deployed with a redaction-oriented target and a lower threshold, recall increases substantially while precision remains high.

The classifier-aware and classifier-free meta variants are almost tied in quality. The classifier-aware variant has the highest exact-matching F2 by a very small margin, while the classifier-free variant is slightly stronger under partial matching. Because the classifier-free variant avoids Secrets-Sentinel and is more than twice as fast (113 ms versus 261 ms), it is the better engineering choice for the default local prototype. This decision should be revisited if a future external evaluation on SecretBench or a real-world dataset shows a different error profile.

The error analysis in [Section 7.8](#) identifies two remaining patterns. Low-entropy passwords have an average entropy of only 3.29, placing them below the range where entropy-based detection is reliable, they require context-sensitive detection. Long compound values such as credential-bearing URLs are harder because their exact span boundaries are difficult to reproduce under strict exact matching. Both are concrete targets for future data generation and rule design rather than fundamental limitations of the architecture.

8.2 Comparison with the Predecessor Prototype

The most direct point of comparison is the predecessor prototype [\[10\]](#). There, secret detection achieved an overall F1 of 0.88 with precision 0.86 and recall 0.90 on a synthetic test set, but per-category recall ranged from 41.7% on plain API_KEYS to 100% on UUIDs and IPv6 addresses. The variance was driven by two factors. First, regex patterns covered some categories well and others poorly. Second, the spaCy NER models had no training signal for cybersecurity-specific tokens. The present work is expected to lift

the floor of the per-category distribution substantially: the fine-tuned model has explicit training signal for every category in the schema, and the hybrid configuration retains the near-perfect recall of regex on structured tokens while adding the contextual capability that regex lacks.

The per-category results in [Table 36](#) and [Figure 35](#) show that the final hybrid pipeline substantially raises the recall floor compared with the predecessor, but also reveals a different weak point. Provider-style tokens such as `APIKey`, `AWSAccessKey` and `GitHubToken`, which the predecessor struggled with most, reaching only 41.7% recall, now reach near-perfect recall under the hybrid configuration, confirming that the fine-tuned GLiNER2 adapter adds the contextual signal that pure regex lacked. At the same time, the new weakest category is `RedisURL`, with an exact matching recall of only 27.9%. This is a different failure mode from the predecessor: rather than missing structured tokens, the final pipeline struggles with long credential-bearing URLs whose exact span boundaries are difficult to reproduce. The floor of the per-category distribution has therefore shifted from structured API key formats to compound URL values, which points to concrete targets for future data generation and rule design rather than a fundamental regression.

8.3 Comparison with Related Work

The closest related work is Rahman et al. [1], who report strong binary-classification results on SecretBench using a fine-tuned LLaMA-3.1 8B model. Their setting is not directly comparable to this thesis. First, the present work reports its final pipeline result on a synthetic test set because SecretBench access was not received in time. Second, this thesis evaluates a local redaction pipeline with span prediction, candidate merging, and final redaction decisions rather than only a remote large-model classification setup. Third, the deployed model stack is intended to run locally, whereas an 8B language model introduces substantially higher hardware and deployment requirements.

KEYSENTINEL [2] and industrial scanners such as Gitleaks [5] focus strongly on large rule catalogs and filtering layers. This thesis follows a different design. The regex catalog is deliberately kept small and conservative, while the broad long-tail decision is delegated to GLiNER2 and the candidate-level meta-classifier. The final results support this direction under the synthetic benchmark: the regex-only baseline is useful and interpretable, but the final hybrid pipeline achieves a much higher exact-matching F1-score.

The comparison also highlights the main limitation of the present thesis. Without a completed SecretBench evaluation, the results cannot yet demonstrate that the synthetic benchmark performance transfers to real-world repositories. The contribution is therefore best understood as a local prototype and controlled synthetic evaluation of a hybrid architecture, not as a final state-of-the-art claim on SecretBench.

8.4 Engineering Trade-offs

Building the prototype as a Tauri application instead of, for example, a containerized web application is a deliberate trade-off. The Tauri approach yields a small installer (a few megabytes for the shell, plus the model weights) and uses the operating system’s native webview, but it introduces a non-trivial build pipeline (Rust toolchain, Python freeze, PyInstaller, sidecar registration). The alternative of shipping a Docker container would be simpler to build but would exclude users on Windows without WSL and would require Docker Desktop, which is not always available in restricted enterprise environments. For our use case (a developer laptop) the Tauri approach fits better, and we accept the build complexity in exchange.

A second trade-off is the choice to bundle the model weights with the installer rather than downloading them on first start. Bundling makes the installer larger but it preserves the offline guarantee: a fresh installation works without any network access. We consider this a non-negotiable property given the key idea behind the use-case of the prototype.

8.5 Research Trade-offs

A central research trade-off concerns the use of synthetic data during development and SecretBench as an external validation dataset. Because SecretBench access was not received in time, the pipeline, training procedure, and comparisons were developed using synthetically generated software artifacts with annotated secrets. This made the project executable within the available timeframe and enabled exact span-level ground truth, controlled secret-category coverage, and reproducible training.

The trade-off is external validity. Synthetic artifacts may contain generator-specific patterns, simplified contexts, or distributions that differ from real repositories and developer workflows. The Jaccard similarity analysis reduces some of this risk, but it cannot replace external validation on real-world data. SecretBench therefore remains important as the intended external benchmark, not as the development dataset used for the reported synthetic results.

A second trade-off is the choice to optimize the final pipeline for a local desktop use case. The selected Meta only configuration avoids Secrets-Sentinel in the default path because it is slightly more accurate in the final benchmark and substantially faster than the classifier-aware meta variant. This choice improves deployability and runtime, but it may also remove a signal that could help on a different external dataset. The decision should therefore be revisited if SecretBench or a private enterprise dataset shows a different error profile.

8.6 Threats to Validity

We discuss the four standard threats to validity in software engineering research [37].

Construct validity. The primary objective of the prototype is redaction: a secret should no longer remain visible in the processed output. The primary evaluation therefore focuses on value-level detection and redaction success rather than on fine-grained label correctness. F2 is the primary aggregate metric, with recall receiving particular attention because a false negative leaves a sensitive value exposed, precision and F1 are also reported. Exact matching is the primary criterion and partial matching is reported separately, because a partially redacted secret may still be security-relevant.

Internal validity. The pipeline was developed and refined using synthetically generated software artifacts with annotated secrets. This provides controlled ground truth and repeatable experiments, but it also creates the risk that model or pipeline decisions are influenced by recurring generation patterns rather than by real-world secret-detection characteristics. To reduce this risk, training, validation and test data were separated before final model selection, and identical or near-identical generated values and templates do not occur across the splits.

The final evaluation originally planned to use SecretBench as an external benchmark. Because access was not received in time, this validation step could not be performed. Consequently, the reported final performance may still be influenced by properties of the synthetic generation process, and development decisions such as model selection, threshold tuning, regex-rule refinement, or classifier-feature selection were all made using synthetic data.

External validity. The synthetic dataset supports controlled development, but it cannot fully represent the variety, noise, and contextual structures found in real software artifacts. In particular, generated configuration files, logs, or code snippets contain regularities that differ from artifacts created in real development environments and are only a limited degree representative of artifacts created in real development environments.

SecretBench would strengthen external validity because it contains labeled secret occurrences extracted from public GitHub repositories. Nevertheless, SecretBench also represents a specific domain: publicly available open-source repositories. Secret formats, naming conventions, logging practices, and surround-

ing context in private enterprise repositories or internal developer workflows may differ. Consequently, even a future successful evaluation on SecretBench would not establish general performance for all real-world deployment contexts.

The prototype use case introduces an additional limitation to external validity. The application targets local redaction of copied code fragments, configuration excerpts, logs, and general texts before they are shared with an external service. This differs from repository-wide secret scanning, continuous integration checks, or historical commit analysis. Conclusions from the thesis should therefore be limited to the evaluated local redaction workflow.

Conclusion validity. The final conclusions depend on whether the reported differences between rule-based baselines, model-based approaches, and the hybrid pipeline are sufficiently robust. All reported configurations are therefore evaluated using the same matching rules and the same fixed synthetic evaluation datasets. Since no SecretBench run is included, the conclusion is limited to this controlled setting and does not claim benchmark-level performance on public repository data.

To support reproducibility, the final experiment configurations, dataset preprocessing steps, model identifiers, thresholds and result exports are versioned and documented ([Section 6.16](#)).

8.7 Possible Use Case

A practical use case ([Figure 38](#)) for the prototype is local redaction before the content is shared with an external service. During their daily work, developers frequently copy configuration snippets, log fragments, source-code excerpts, error messages, or deployment information into external communication channels or AI-based assistants. Such content may unintentionally include passwords, access tokens, connection strings, webhook URLs, or other credentials.

The developed prototype addresses this workflow by introducing a local redaction step immediately before the content leaves the user’s device. In the regular application workflow, a user can paste copied text into the desktop application, initiate redaction, review the processed output, and copy only the redacted version to the external destination. This provides a simple and understandable interaction model while keeping the detected secrets local.

A more efficient workflow is enabled through the keyboard shortcut `Ctrl+Shift+R`. In this scenario, a user copies a text fragment as usual, triggers local redaction through the shortcut, and receives the redacted version back into the clipboard. The processed content can then be pasted into an external chatbot, ticket system, support channel, or other service without having to interact with the UI of the prototype. Compared with the normal copy-and-paste workflow, this adds only one additional user action while potentially preventing sensitive credentials from being disclosed.

This use case is not limited to large organizations with dedicated security infrastructure. Because the prototype is designed for local, CPU-based execution without dedicated hardware requirements, it could also be applicable to small organizations, small development teams, freelancers, or private users. The absence of mandatory server infrastructure lowers the technical adoption barrier and supports use in environments where a lightweight local safeguard is preferable to a centralized scanning system.

From an organizational perspective, the prototype could act as an additional protective measure in developer workflows. It may help reduce accidental secret exposure when employees use external assistants for debugging, code explanation, configuration support, or communication.

However, the prototype must not be interpreted as a guarantee that shared content is free of secrets. Secret-detection systems can miss sensitive values, especially when formats are unknown, contexts are ambiguous, or values resemble ordinary text. Therefore, manual review remains an essential part of the intended workflow. The application supports the user by reducing the likelihood of accidental disclosure, but it does not replace user responsibility or broader organizational security controls.

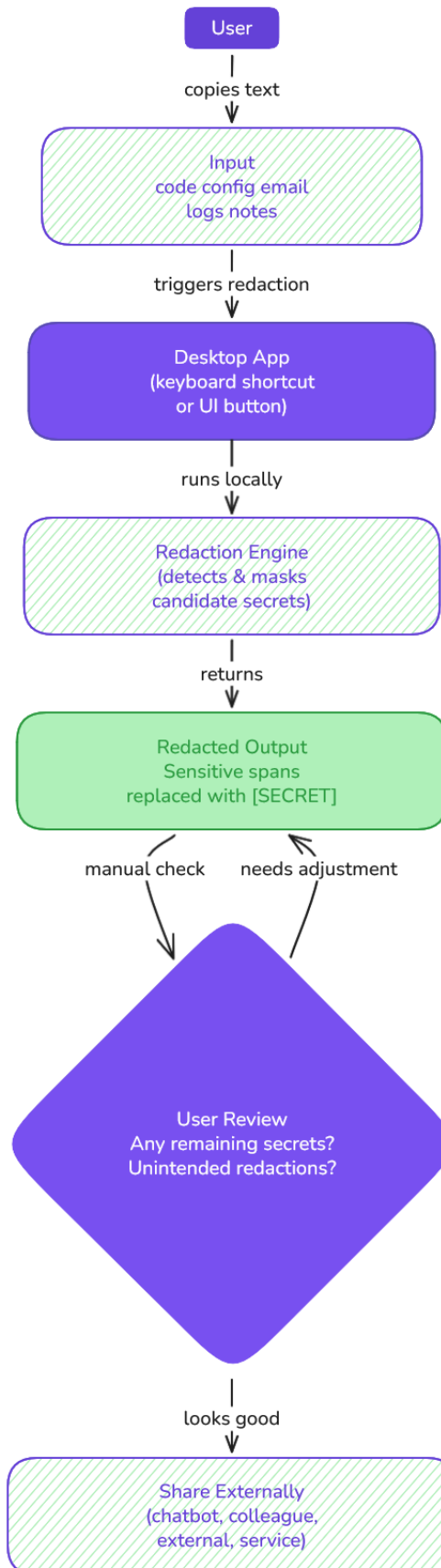


Figure 38: A possible operational workflow.

9 Conclusion

This thesis developed a local secret-redaction prototype that combines regex and entropy candidate generation, a fine-tuned GLiNER2 span detector, candidate merging, and an overlap-aware LightGBM meta-classifier. The prototype is packaged as a Tauri desktop application and is designed to process technical text locally before users share it with external AI assistants, communication tools or code-review systems.

The work was motivated by the gap identified in the predecessor project [10]. Redaction of personally identifiable information worked well there, but secret-specific values such as API keys and passwords remained difficult. This thesis therefore focused on secrets rather than PII and treated redaction as the primary objective. As a consequence, recall was considered especially important, and the final evaluation uses F2 as the main aggregate metric because it weights recall more strongly than precision.

9.1 Answer to the Research Question

The main research question asked whether a hybrid local pipeline combining rule-based, heuristic and learned components improves secret redaction compared with any individual approach alone. The results confirm this under controlled synthetic conditions, as detailed in [Section 8.1](#). The decisive factor is the separation of broad candidate generation from a learned final decision layer: the combination of regex, entropy and GLiNER2 provides the necessary recall coverage, and the LightGBM meta-classifier turns that coverage into a precise and practical redaction decision.

Compared with the regex-only baseline, the final pipeline improves exact-matching F1 from 0.6156 to 0.9492. Compared with the unfiltered candidate union, it reduces false positives from 21626 to 303 while preserving recall at 0.9486. The F2 progression across all evaluated configurations, shown in [Figure 31](#), makes this cumulative improvement visible at a glance.

The result is not yet externally validated: SecretBench or another labeled real-world dataset remains necessary before making performance claims beyond the synthetic benchmark, and this is the most important direction for future work ([Section 9.2](#)).

9.2 Future Work

Several directions could extend the present work. They are listed in rough order of expected impact and feasibility.

External evaluation on real-world data. The most important next step is to evaluate the frozen final pipeline on SecretBench or another labeled real-world repository dataset. This would test whether the synthetic-dataset findings transfer to labeled real-world repository data.

Local runtime and packaging evaluation. The benchmark reports runtime in the Kaggle evaluation environment. A separate local benchmark on the packaged Tauri prototype is needed to measure startup time, CPU inference latency, memory usage, and installer size on realistic user devices.

File-level scanning. The use case targets one pasted or written text at a time. A file-level and or folder-level scanner which can process mutiple files over time could be a fitting complement, although it brings new challenges.

9.3 Closing Remarks

The project shows that a useful local secret-redaction workflow requires more than a single detector. Regex rules, entropy heuristics and neural span detection each contribute useful signals, but they also produce different error profiles. The strongest result in this thesis comes from treating these stages as candidate generators and then applying a small learned decision layer to decide what should actually be redacted.

Two further lessons are worth noting. First, the regex catalog turned out to be more valuable as a candidate generator than as a primary detector: once a model is in the loop, a small, carefully chosen catalog combined with a good model dominates a large catalog combined with a weak classifier, and every extra pattern mostly adds maintenance cost. Second, dataset availability proved to be not merely a technical dependency but a central factor in the validity and scope of the conclusions. The unavailable SecretBench access made synthetic generation necessary, this enabled exact ground-truth spans, controllable categories and fast iteration, but it also bounds the claims to controlled synthetic conditions and automatically introduces overfitting based on the data.

The resulting prototype is not a finished security product. Its final numbers are based on a synthetic benchmark, and external validation remains necessary. Nevertheless, the thesis demonstrates a practical path toward local redaction: broad detection, recall-oriented threshold selection, a learned final decision layer, and an offline desktop workflow that lets users reduce the risk of accidentally sharing secrets with external services.

9.3 Bibliography

- [1] M. N. Rahman, S. Ahmed, Z. Wahab, S. M. Sohan, and R. Shahriyar, “Secret Breach Detection in Source Code with Large Language Models.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2504.18784>
- [2] J. Zhou, Z. Zhang, L. Ying, H. Chai, J. Cao, and H. Duan, “Hey, Your Secrets Leaked! Detecting and Characterizing Secret Leakage in the Wild,” in *2025 IEEE Symposium on Security and Privacy (SP)*, May 2025, pp. 449–467. doi: [10.1109/SP61157.2025.00122](https://doi.org/10.1109/SP61157.2025.00122).
- [3] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, “Detecting and Mitigating Secret-Key Leaks in Source Code Repositories,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 396–400. doi: [10.1109/MSR.2015.48](https://doi.org/10.1109/MSR.2015.48).
- [4] Truffle Security, “TruffleHog: Find, verify, and analyze leaked credentials.” Accessed: June 01, 2026. [Online]. Available: <https://github.com/trufflesecurity/trufflehog>
- [5] Z. Rice, “gitleaks/gitleaks.” Accessed: May 23, 2026. [Online]. Available: <https://github.com/gitleaks/gitleaks>
- [6] GitGuardian, “GitGuardian: Secrets Security and NHI Governance.” Accessed: June 01, 2026. [Online]. Available: <https://www.gitguardian.com/>
- [7] GitHub, Inc., “GitHub: The complete developer platform.” Accessed: June 01, 2026. [Online]. Available: <https://github.com/>
- [8] U. Zaratiana, N. Tomeh, P. Holat, and T. Charnois, “GLiNER: Generalist Model for Named Entity Recognition using Bidirectional Transformer.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2311.08526>
- [9] U. Zaratiana, G. Pasternak, O. Boyd, G. Hurn-Maloney, and A. Lewis, “GLiNER2: An Efficient Multi-Task Information Extraction System with Schema-Driven Interface.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2507.18546>
- [10] E. Kaiser and N. Heiniger, “Sensitive Data AI Redactor: A Local Pipeline for PII and Secret Detection in Text and Speech,” *Forschungsarbeit*, 2025. [Online]. Available: <https://eprints.ost.ch/id/eprint/1369/1/HS%202025%202026-SA-EP-Kaiser-Heiniger-KI-gest%C3%BCtzt%20Erkennung%20von%20sensiblen%20Informationen%20in%20Source.pdf>
- [11] SYSTRAN, “faster-whisper: Faster Whisper transcription with CTranslate2.” Accessed: June 01, 2026. [Online]. Available: <https://github.com/SYSTRAN/faster-whisper>
- [12] Microsoft, “Presidio: Data protection and de-identification SDK.” Accessed: June 01, 2026. [Online]. Available: <https://github.com/microsoft/presidio>
- [13] S. K. Basak, L. Neil, B. Reaves, and L. Williams, “SecretBench: A Dataset of Software Secrets.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2303.06729>
- [14] “Supported secret scanning patterns.” Accessed: May 23, 2026. [Online]. Available: <https://docs.github.com/en/code-security/reference/secret-security/supported-secret-scanning-patterns>
- [15] S. K. Basak, J. Cox, B. Reaves, and L. Williams, “A Comparative Study of Software Secrets Reporting by Secret Detection Tools,” 2023, [Online]. Available: <https://arxiv.org/abs/2307.00714>
- [16] “Yelp/detect-secrets.” Accessed: May 23, 2026. [Online]. Available: <https://github.com/Yelp/detect-secrets>

- [17] F. Soltaniani and M. Ghafari, “From Data Leak to Secret Misses: The Impact of Data Leakage on Secret Detection Models.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2601.22946>
- [18] J. E. Hopcroft, “Introduction to Automata Theory, Languages, and Computation: | Guide books | ACM Digital Library.” Accessed: May 23, 2026. [Online]. Available: <https://dl.acm.org/doi/book/10.5555/1454320>
- [19] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948, doi: [10.1002/j.1538-7305.1948.tb01338.x](https://doi.org/10.1002/j.1538-7305.1948.tb01338.x).
- [20] T. M. Cover and J. A. Thomas, *Elements of information theory*, Second edition. Hoboken, NJ: Wiley-Interscience, 2006. doi: [10.1002/047174882X](https://doi.org/10.1002/047174882X).
- [21] E. F. Tjong Kim Sang and F. De Meulder, “Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition,” in *Proceedings of the Seventh Conference on Natural Language Learning (CoNLL-2003)*, 2003, pp. 142–147. [Online]. Available: <https://aclanthology.org/W03-0419>
- [22] A. Vaswani *et al.*, “Attention Is All You Need.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/1810.04805>
- [24] J. Howard and S. Ruder, “Universal Language Model Fine-tuning for Text Classification.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/1801.06146>
- [25] E. J. Hu *et al.*, “LoRA: Low-Rank Adaptation of Large Language Models.” Accessed: May 23, 2026. [Online]. Available: <http://arxiv.org/abs/2106.09685>
- [26] “hypn05/secrets-sentinel · Hugging Face.” Accessed: May 23, 2026. [Online]. Available: <https://huggingface.co/hypn05/secrets-sentinel>
- [27] S. Brown, “The C4 model for visualising software architecture.” Accessed: June 01, 2026. [Online]. Available: <https://c4model.com/>
- [28] “Tauri 2.0.” Accessed: May 23, 2026. [Online]. Available: <https://v2.tauri.app/>
- [29] FastAPI, “FastAPI documentation.” Accessed: June 01, 2026. [Online]. Available: <https://fastapi.tiangolo.com/>
- [30] PyInstaller, “PyInstaller Manual.” Accessed: June 01, 2026. [Online]. Available: <https://www.pyinstaller.org/>
- [31] G. Ke *et al.*, “LightGBM: A Highly Efficient Gradient Boosting Decision Tree,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2017. Accessed: May 23, 2026. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html
- [32] “Étude comparative de la distribution florale dans une portion des Alpes et du Jura,” *Bulletin de la Société Vaudoise des Sciences Naturelles*, vol. 37, no. 142, 1901, Accessed: May 27, 2026. [Online]. Available: <https://www.e-periodica.ch/digbib/view?pid=bsv-002%3A1901%3A37%3A%3A579>
- [33] OpenAI, “OpenAI Privacy Filter: Open-weight bidirectional token classifier for privacy-sensitive span detection.” Accessed: June 01, 2026. [Online]. Available: <https://huggingface.co/openai/privacy-filter>
- [34] N. Gupta, “DeepPass2: Fine-tuned XLM-RoBERTa for Secret Detection.” Accessed: June 01, 2026. [Online]. Available: <https://huggingface.co/gneeraj/deeppass2-bert>

- [35] NVIDIA, “nvidia/gliner-P11 · Hugging Face.” Accessed: May 23, 2026. [Online]. Available: <https://huggingface.co/nvidia/gliner-P11>
- [36] Google LLC, “Kaggle: The world’s AI proving ground.” Accessed: June 01, 2026. [Online]. Available: <https://www.kaggle.com/>
- [37] R. Per and M. Höst, “Guidelines for conducting and reporting case study research in software engineering | Empirical Software Engineering | Springer Nature Link.” Accessed: May 23, 2026. [Online]. Available: <https://link.springer.com/article/10.1007/s10664-008-9102-8>
- [38] OpenAI, “ChatGPT.” [Online]. Available: <https://chat.openai.com/>
- [39] DeepL SE, “DeepL Translator.” [Online]. Available: <https://www.deepl.com/>
- [40] Google LLC, “Google Translate.” [Online]. Available: <https://translate.google.com/>
- [41] Perplexity AI, “Perplexity.” [Online]. Available: <https://www.perplexity.ai/>
- [42] Python Software Foundation, “Python Programming Language.” Accessed: June 01, 2026. [Online]. Available: <https://www.python.org/>
- [43] Microsoft, “Visual Studio Code: The open source AI code editor.” Accessed: June 01, 2026. [Online]. Available: <https://code.visualstudio.com/>
- [44] JGraph Ltd, “draw.io: Flowchart maker and online diagram software.” Accessed: June 01, 2026. [Online]. Available: <https://www.drawio.com/>
- [45] Excalidraw s.r.o., “Excalidraw: Virtual whiteboard for sketching hand-drawn like diagrams.” Accessed: June 01, 2026. [Online]. Available: <https://excalidraw.com/>
- [46] Corporation for Digital Scholarship, “Zotero: Your personal research assistant.” Accessed: June 01, 2026. [Online]. Available: <https://www.zotero.org/>
- [47] Microsoft, “Microsoft 365: Productivity apps and cloud services.” Accessed: June 01, 2026. [Online]. Available: <https://www.microsoft365.com/>
- [48] Typst GmbH, “Typst: A new markup-based typesetting system.” Accessed: June 01, 2026. [Online]. Available: <https://typst.app/>

9.3 List of Abbreviations

AI	Artificial Intelligence
AP	Average Precision
API	Application Programming Interface
AWS	Amazon Web Services
BERT	Bidirectional Encoder Representations from Transformers
CI/CD	Continuous Integration / Continuous Delivery
CPU	Central Processing Unit
F1	Harmonic mean of precision and recall
F2	F-measure with $\beta = 2$ (recall-weighted)
FN	False Negative
FP	False Positive
GLiNER	Generalist and Lightweight Model for Named Entity Recognition
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSONL	JSON Lines
JWT	JSON Web Token
LLM	Large Language Model
LoRA	Low-Rank Adaptation
LR	Learning Rate
ML	Machine Learning
NER	Named Entity Recognition
PEM	Privacy-Enhanced Mail (key encoding)
PGP	Pretty Good Privacy
PII	Personally Identifiable Information
RQ	Research Question
SHA	Secure Hash Algorithm
SSH	Secure Shell
TP	True Positive
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VCS	Version Control System

9.3 Glossary

Candidate	A substring of an input text that has been flagged by a detector (regex, entropy or model) as potentially being a secret. A candidate is not yet a final detection, meaning it has to pass classification and de-duplication first.
Confidence threshold	The minimum probability a model must assign to a span before it is accepted as a positive prediction. Lower thresholds favor recall, higher thresholds favor precision.
De-duplication	The process of removing duplicate or near-duplicate samples from a dataset, or of merging overlapping detections that refer to the same secret value.
Entropy	Shannon entropy, a measure of the average information content of a string. Random-looking, high-entropy strings are typical of cryptographic keys, whereas natural text has significantly lower entropy.
Exact matching	An evaluation criterion under which a prediction counts as correct only if its character boundaries match the annotated secret span exactly. It is the strict primary criterion used in this thesis.
Fine-tuning	The process of continuing the training of a pretrained model on a domain-specific dataset, typically with a lower learning rate, in order to adapt the representations to a new task without retraining from scratch.
GLiNER	A bidirectional encoder NER architecture that performs zero-shot extraction by matching entity-type prompts to text spans in a shared latent space. It runs efficiently on CPU and contains fewer than 500M parameters.
Hard negative	A value that resembles a secret (for example a sample token, placeholder password or UUID) but is not a real credential. Hard negatives are inserted into the dataset to test whether a detector can avoid over-redaction.
Hybrid pipeline	A detection pipeline that combines several signal sources, in this work regex, entropy and a transformer-based classifier, to leverage the strengths and compensate the weaknesses of each individual approach.
Meta-classifier	The final learned decision layer of the pipeline (a LightGBM model) that takes features describing a merged candidate and decides whether it should be redacted. It separates broad candidate generation from the final keep/drop decision.
Partial matching	A secondary evaluation criterion under which a prediction counts as correct if it overlaps the annotated secret span, optionally above a coverage threshold. It describes incomplete but still useful detections.
Redaction	The act of replacing a sensitive substring with a placeholder so that the surrounding text remains readable but the secret value is no longer visible.
Regular expression	A formal pattern over an alphabet that defines a set of strings. Regular expressions are deterministic, fast and well-suited for structured tokens such as AWS keys, but they cannot reason about context.
SecretBench	A public benchmark dataset constructed from real-world GitHub repositories that contains roughly 97000 candidate secrets and 15000 manually verified true secrets across many programming languages and file types.

Secrets-Sentinel	An optional model-based candidate-level classifier that estimates whether an already extracted candidate is a real secret. It is used as an additional feature signal, not as the final decision, and is excluded from the default pipeline for runtime reasons.
Span	A contiguous range of characters in a text, typically described by a start and end offset. Spans are the basic prediction unit of GLiNER.
Tauri	A framework that packages a web-based front-end together with a Rust core into a small native desktop application. It is used as the runtime for the prototype.
Token	Either an authentication artifact (for example an OAuth bearer token), or, in NLP, a unit of text produced by a tokenizer such as a subword or byte-pair encoding piece. In this work, both meanings are used and distinguished from each other by the context.
True secret	A span annotated as a genuine secret in the evaluation dataset, as opposed to a hard negative, dummy value or other secret-like string that does not represent a real, functional credential.

9.3 List of Figures

Figure 1	Architecture of the predecessor prototype.	2
Figure 2	Shannon entropy distribution of secrets and non-secret strings in the dataset.	11
Figure 3	Influence diagram of the evaluation strategy.	13
Figure 4	System context diagram of the redaction prototype.	16
Figure 5	Container diagram of the redaction prototype.	17
Figure 6	Tauri application architecture and build pipeline overview.	17
Figure 7	The Sensai prototype during a redaction session.	18
Figure 8	C4 component diagram of the backend redaction engine.	19
Figure 9	Rule-based redaction pipeline using regex and entropy	22
Figure 10	NER-based redaction pipeline using GLiNER2.	23
Figure 11	Hybrid candidate generation using rule-based and NER-based detection.	23
Figure 12	Final hybrid pipeline with LightGBM meta-classifier and masking.	24
Figure 13	Runtime process flow of the hybrid redaction pipeline.	25
Figure 14	Distribution of the regex catalog.	26
Figure 15	Distribution of annotated secret spans by original secret type.	31
Figure 16	Length and entropy distributions of generated artifacts and secret spans.	32
Figure 17	Normalized token-set Jaccard similarity across dataset splits.	33
Figure 18	F1 and F2 of performed GLiNER2 LoRA configuration experiments.	39
Figure 19	Exact-matching F2 of regex-only, entropy-only and combined variants.	44
Figure 20	Partial-matching F2 of regex-only, entropy-only and combined variants.	45
Figure 21	Exact matching F2 of the zero-shot model comparison.	46
Figure 22	Exact matching F2 versus runtime of zero-shot model.	47
Figure 23	Partial F2 of the zero-shot model comparison.	48
Figure 24	Training and validation loss of the GLiNER2 LoRA fine-tuning run.	49
Figure 25	GLiNER2 secret prediction score distribution on the test set.	50
Figure 26	Confusion matrix of the selected LightGBM meta-classifier.	52
Figure 27	Best transformed feature importances of selected LightGBM meta-classifier.	53
Figure 28	LightGBM learning curves of the selected classifier-free meta-classifier.	53
Figure 29	Meta-classifier threshold sweep on the synthetic test set using exact matching. ...	54
Figure 30	Meta-classifier threshold sweep on the synthetic test set using partial matching. .	55
Figure 31	Exact-matching F2 progression across pipeline configurations.	56
Figure 32	Exact-matching precision, recall, F1 & F2 of the hybrid pipeline configurations. ...	57
Figure 33	Stage-wise exact-matching contribution of each candidate stage.	58
Figure 34	F1 and F2 across overlap coverage thresholds of final pipeline.	59
Figure 35	Actual secret types with the lowest exact matching F2 scores.	60
Figure 36	Shannon entropy distribution of false negatives and false positives.	61
Figure 37	Exact-matching metrics across entropy thresholds.	63
Figure 38	A possible operational workflow.	69

9.3 List of Tables

Table 1	Comparison of secret-detection tools and detection strategies.	7
Table 2	Examples of secret types and secret-like values.	9
Table 3	Examples of regular expressions for common secret formats.	10
Table 4	Baseline and hybrid configurations.	15
Table 5	Conceptual design iterations of the secret-redaction pipeline.	22
Table 6	Roles of the synthetic dataset and SecretBench in this thesis.	27
Table 7	Published category distribution of SecretBench.	28
Table 8	Structure of the generated JSONL dataset rows	29
Table 9	Synthetic dataset split sizes and roles.	30
Table 10	Configuration of the regex and entropy baseline experiments	35
Table 11	Scope of the zero-shot model comparison	35
Table 12	Pretrained models compared in zero-shot mode.	36
Table 13	Fine-tuning configuration of the GLiNER2 LoRA adapter.	36
Table 14	Synthetic subsets used for the GLiNER2 LoRA configuration experiments.	37
Table 15	Executed GLiNER2 LoRA configuration experiments.	38
Table 16	Exact-span GLiNER2 LoRA configuration experiment results sorted by test F2.	38
Table 17	Five-fold cross-validation summary of best GLiNER2 LoRA configuration.	39
Table 18	Validation fold balance for the five-fold robustness check.	40
Table 19	Feature families used for candidate-level meta-classifier training.	41
Table 20	Meta-classifier variants and their feature availability assumptions.	42
Table 21	Hybrid pipeline benchmark configurations.	42
Table 22	Shared benchmark settings for the final hybrid pipeline comparison.	43
Table 23	Exact value-level results of the regex and entropy baselines.	44
Table 24	Partial-matching results of the regex and entropy baselines.	45
Table 25	Zero-shot model comparison under exact matching.	46
Table 26	Zero-shot model comparison under partial matching.	47
Table 27	Final GLiNER2 LoRA performance at threshold 0.8.	50
Table 28	Best reported GLiNER2 configuration and its exact value-level metrics.	51
Table 29	Final LightGBM candidate-level meta-classifier results.	52
Table 30	Best criterion of the LightGBM classifier on the synthetic test set.	54
Table 31	Best 90%-overlap criterion of the LightGBM classifier.	55
Table 32	Exact-matching performance of the final hybrid pipeline configurations.	56
Table 33	Partial-matching performance of the final hybrid pipeline configurations.	57
Table 34	Stage-wise exact-matching contribution of the individual candidate stages.	58
Table 35	Exact, partial and thresholded-overlap metrics of selected final pipeline.	59
Table 36	Lowest exact matching F2 secret types with at least 20 target spans.	60
Table 37	Aggregate properties of exact-matching errors.	61
Table 38	Most frequent missed secret types.	62
Table 39	Representative shortened error examples from the selected final pipeline.	62
Table 40	Average runtime and performance of the hybrid pipeline configurations.	64
Table 41	AI-based tools used during the project.	80
Table 42	General tools used during the project.	81

A AI Tool Usage Disclosure

AI-based tools were used during the project as productivity aids for brainstorming, language refinement, translation support, debugging assistance and documentation review. They were not used as authoritative sources. All technical decisions, claims, citations, implementation details and final text remain the responsibility of the authors. [Table 41](#) lists the tools that were used.

ChatGPT [38]	Used for brainstorming, structuring chapters, language refinement and English formulation support. Supplementary used for debugging assistance, frontend styling ideas and reviewing wording consistency.
DeepL [39]	Used for translation between German and English and for alternative wording suggestions.
Google Translate [40]	Occasionally used for quick translation checks and comparison with other outputs to ensure correctness and consistency.
Perplexity [41]	Used for research overviews, paper discovery and summarising related work during the literature review phase.

Table 41: AI-based tools used during the project.

B Tooling

The tools used during the project are listed in Table 42. We focused on tools that are widely available, free for academic use and that support reproducible workflows. More specific AI-related tools are listed separately in Table 41 in the appendix.

GitHub [7]	Used for version control and collaboration during development.
Python [42]	Primary implementation language for the generator, training, evaluation and benchmark scripts.
Visual Studio Code [43]	Used as the primary IDE for development, debugging and refactoring of the code.
Kaggle [36]	Used for experiments and benchmarks with access to stronger CPU/GPU hardware than the local development machine.
draw.io [44]	Used to create fast architecture and process sketches, especially the C4 and pipeline diagrams.
Excalidraw [45]	Used for quick sketches, early visual concepts and informal diagrams.
Zotero [46]	Used to manage bibliography entries, collect source metadata and export BibTeX references.
Microsoft Teams [47]	Used for communication, supervisor meetings and project coordination.
Microsoft Excel [47]	Used for inspecting exported metrics, comparing benchmark results and preparing small result tables.
Microsoft Word [47]	Used for occasional text review, spell checking and comparison with institutional formatting expectations.
Typst [48]	Used as the main typesetting system for writing and compiling the thesis document.

Table 42: General tools used during the project.

End of document.