Yeti: A Software Suite for Detecting and Analyzing Code Plagiarism at Hackathon Competitions

A Major Qualifying Project submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for the degree of Bachelor of Science

by
Kendall Goto

Date:
27 Apr 2022

Report Submitted to:

Professor Jonathan Weinstock
Worcester Polytechnic Institute

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.
**ABSTRACT**

Hackathon events are run frequently by high schools, colleges, and corporations in order to utilize a competitive environment to inspire rapid product ideation and development within a community. However, since the start of the COVID-19 pandemic, many hackathon events have moved to an online environment, creating widespread issues of fraudulent, stolen, or reused projects being submitted to events seeking to exploit their monetary or in-kind rewards. “Yeti” is a proposed software suite designed to be utilized by hackathon event organizers to reduce the risk of fraudulent work by tracking competitor submissions end-to-end, from initial creation to final submission with intermediate checkpoints. In this paper, the requirements, development, and testing of an early Yeti proof of concept is proposed in detail to serve as a basis for the eventual production of Yeti as a commercial software product.
**TABLE OF CONTENTS**

Abstract

Table of Contents

Background

   Hackathon Events
   Issues in the Hackathon Space
   Hackathon Stakeholders
   Pinnacle

Proposal

   Product Summary
   Product Features
      User-level Interface
      Command-line Interface
      Yeti Core
      Reporting
   Product Use Case Study
      Before the Event
      During the Event
      During Event Judging
      After the Event

   Go To Market Strategy

Implementation

   Existing Work
      Academic Plagiarism Detection
      Industry Interview Plagiarism Detection
      Commercial Plagiarism Detection
   Technical Goals

   Architecture Design
      Yeti CLI
      Yeti DB
      Yeti API
      Yeti Git
   Yeti Engine
      Pipeline Stage: Collecting
      Pipeline Stage: Unifying
      Pipeline Stage: Slicing
      Pipeline Stage: Matching
Pipeline Stage: Reporting

**Results and Testing**
- Functional Testing (Alpha)
  - Test Requirements
  - Test Plan
  - Test Results
- Pre-Production Testing (Beta)
  - Test Requirements
  - Test Plan
- Field Trial Testing (Release Candidate)
  - Test Requirements
  - Test Plan
- Production Testing (GA)
  - Test Requirements
  - Test Plan

**Commentary and Conclusions**
- Plagiarism Deterrence and Plagiarism Filtering
- Protecting Open Source Algorithms from Abuse
- Future Work
- Concluding Thoughts

**References**

**Appendix**
- Appendix I: Proof of Concept Demonstration Sequence Diagram
- Appendix II: Software Architecture Stack
- Appendix III: The Yeti Pipeline (Marketing Imagery)
BACKGROUND

The work in this paper is primarily centered on an understanding of the “hackathon” scene. A large amount of the business requirements and proposals stem from a direct understanding of hackathons attendees and organizers. Through this paper, it will be demonstrated how Yeti is designed for and by hackathon experts as the first software package of its kind.

Hackathon Events

Hackathons take many different forms. Broadly, hackathons are events in which engineers, designers, and marketers come together to “hack” together a product in a short, concentrated time frame. These events can be run by organizations looking to challenge their staff or local community to create products. Occasionally, organizations may use hackathons as an idea ideation strategy, challenging participants to rapidly create and test a large number of product ideas. Hackathons may focus on a certain topic, such as financial technology or cryptocurrencies, or may be broad with no specific topic. The submissions may take different forms, depending on the focus and participant skill set, such as a marketing pitch, a demo-ready application, or design wireframes. Attendees sometimes work individually, or in teams depending on the scope of the event, and the length of time can vary broadly, from 12 hours to a week. The event can either occur in person or virtually, or a combination of both.

Collegiate and high school hackathons are typically limited to only student-age attendees. They tend to occur on a school’s campus or otherwise be connected to / affiliated with a university or high school. Most events occur over the course of a weekend, with participants working, sleeping, and eating on-site to build their product in the limited time frame. Collegiate and high school events are typically organized by students at their respective schools and funded by sponsor corporations.

Issues in the Hackathon Space

Cheating in hackathons: Broadly, cheating in a hackathon involves creating unoriginal work, such as stealing projects or ideas, and passing them off as one’s own. One common form of cheating is to steal code, designs, or product implementation details from open source projects but indicating to judges that they are novel concepts produced by a team. Another common form of cheating is preparing for an event ahead of time by writing code or creating final designs prior to the event’s limited time frame.
Lack of oversight: Most hackathons have a proportionally small group of organizers compared to attendees - often a 1:10 ratio of organizers to attendees or worse. This means organizers are not able to appropriately oversee and understand each team’s work as it develops over the course of the event, and unable to address misunderstandings or nefarious behavior.

Inability to enforce rules: Oftentimes, hackathon organizers create idealistic rules but have little means to enforce these rules with limited resources and little insight into individual project’s codebase, participant backgrounds, et cetera.

Judging exploitation: At the conclusion of an event, judges have a limited amount of time to see each project in order to understand it and judge it to determine a final winner. As such, participants are able to exaggerate the complexity and completion of their work to be judged in higher esteem.

Hackathon Stakeholders

Hackathons involve numerous parties with very different perspectives that are impacted by the aforementioned issues in different ways.

Organizers: Staff members of the hackathon organizing organization that work before, during, and after a hackathon event to oversee the whole event. As the responsible party for running the event, organizers need to find their own solutions towards resolving the aforementioned issues in order to create an impactful event.

Attendees: Guests that attend hackathons and make code contributions in hope of creating a winning project and being awarded with a prize. Attendees are impacted by the event’s overall integrity in order to feel motivated to contribute in a fair, competitive environment.

Judges: Industry professionals that review attendee projects and pick overall hackathon winners. Judges want to ensure that the projects they review are genuine and avoid tarnishing their personal reputation by unintentionally lauding a fraudulent or exaggerated project.

Sponsors: Companies or individuals that provide funding for hackathon events, often in exchange for marketing promotions or access to attendee profiles for recruitment purposes. Sponsors are interested in protecting their brand by associating themselves with competitive and non-controversial events.

Viewers: Third party spectators who watch the event, either from the sponsor or attendee perspective, out of a wide variety of interests. Viewers are interested in seeing a fair competition that they would be interested in contributing to in the future.
The work in this paper, and the development of Yeti, is guided by Pinnacle, organizers of a premier collegiate hackathon event. Pinnacle’s goal is to push the boundary within the hackathon scene, and create the most competitive event possible. They accomplish this task by creating an exclusive, invitational event that brings the first place contestants of notable collegiate events together for an intense weekend of competition. First place winners from events operated by the University of Waterloo, Stanford, UC Berkeley, and 50 other prestigious institutions are invited to compete head-to-head while recruiters and industry executives watch their work.

Due to the serious nature of Pinnacle’s competition, the organization seeks to create novel software to facilitate the legitimacy and authenticity of the competition. This software is innovative as a hackathon scene first, and aims to demonstrate the ability of combining various strategies to greatly reduce incidents of plagiarism, illegitimate submissions, and nefarious behavior at Pinnacle’s flagship event as well as at Pinnacle’s partnered feeder events.
PROPOSAL

In order to address the aforementioned issues, the development of “Yeti” is proposed, a software package aimed to facilitate the hackathon process end-to-end.

Product Summary

Yeti will be presented as an all-in-one package that delivers a web interface, command-line interface, and performs background tasks to mitigate observed hackathon challenges. The primary interface is a web-based portal that can be utilized by all stakeholders, although with different perspectives. Attendees, sponsors, and viewers primarily access the user-level interface. Organizers and judges access an admin-level interface. The simplified flow of data through the program begins with the user-level interface, feeds through Yeti’s “pipeline”, then is output onto the admin-level interface.

Yeti’s core feature is to create an anti-cheat “pipeline” for use by hackathon organizers and judges. As attendees develop their projects, their individual code contributions will be funneled through the pipeline and analyzed for potential code plagiarism or other behavior against the rules of the event. For the first time in the collegiate hackathon scene’s history, organizers will be able to follow along with the development of each hackathon project, assisted by Yeti’s intelligent pipeline, in order to better manage the event and judging liability.

Product Features

User-level Interface

The core user-facing interface will be designed to be familiar for both organizers and attendees, to allow for easy onboarding. Existing services, like Github and Gitlab, provide a standardized organization for code-centric UIs, and Yeti’s core UI will follow suit.
The “YetiGit” interface will center on a project’s page featuring its code and textual writeup, and commit-level changes will be visible in the UI. Additionally, Yeti core remarks will be generated per-commit, and visually shown. At a glance, it will be clearly evident that a codebase has been entirely scanned by Yeti’s core. Full reports will be visible in this dashboard too, though their access and detail may be limited to certain users.

Via the YetiGit site, organizers will be able to control the overall event settings and attendees will be able to control their team settings and individual repository settings. Access and identification to the overall system can be controlled either by a central ID server, use of external SSO, et cetera.

This YetiGit interface is designed to be the primary visual interface for the Yeti ecosystem, but is not a mandatory facet. That is to say, Yeti’s core is separately constructed from the YetiGit interface, so they are not required to be used together in all installations.

**Command-line Interface**

Since the Yeti platform presents itself as a standard Git server, it can be interfaced directly through the `git` CLI utility. This will provide reasonable control for code and artifact submissions, although certain configurations (repository visibility, release messages, etc) will have to be externally managed, since the Git protocol does not support this type of payload.
An additional CLI may also be created, as a lighter weight alternative to the web interface. This may be a useful tool for automated tasks, and will be an invaluable tool if Yeti is deployed without the YetiGit interface. This may allow for event configuration and setup on the administrative side and allow for project setup on the attendee side.

Yeti Core

This paper is primarily concerned with the implementation of the “heart” of the system, the Yeti Core. This component outlines the majority of Yeti’s functional benefit, and aims to serve the core use case of reducing plagiarism and competitive event liability. While it is a key detail, it is also the intermediate glue that is largely invisible to the end-user on the user-facing and admin-facing sides. The core contains three major components: the Yeti Pipeline, the Yeti API, and the Yeti Database.

The Yeti Pipeline is the processing algorithm to intake code and generate reports. Code is fed in via a variety of sources (predominantly the YetiGit interface) and analyzed via the pipeline. The pipeline’s specific steps will be outlined as part of its software architecture at a later time. Processing results generated by the pipeline will be retained in the Yeti Database for long term archival and lookup.

The Yeti API serves as a primary business logic bridge between the Pipeline, Database, and external sources. It will be the interface used to push data into the pipeline and read reports out, as well as read data directly out of the database as required. This API is required to be a concrete standard, since all non-core Yeti components will be dependent on it for communication and stitching.

The Yeti Database is intentionally abstracted by the Yeti Core in order to maintain flexibility in the underlying architecture. Since multiple Yeti components will be dependent on the database, defining the abstraction layer is crucial to ensuring that the underlying database is resilient, well managed, and adaptable to a given situation. To fulfill the need of the Yeti Database, a standard off-the-shelf DBMS can be used, such as PostgreSQL or MySQL, with a Yeti API provided abstraction layer on top.

Reporting

A core component of the Yeti project will be effectively gathering and presenting the results, for the purposes of event alerts and pre-judging reports. As reports will be generated via the Yeti core, these will need to be parsed and read in via an external component, and fed back to the front-facing UI. For the reports to be effective, they must be generated in a timely fashion and be clear in showing evidence to support the report claim.
An enhanced reporting system should be able to reverse the generation process, in order to follow the “breadcrumbs” and validate the claim itself fully from the source. For instance, if a plagiarized piece of code is submitted, a thorough report should include the original code, its source and author, and original license.

**Product Use Case Study**

This section will detail the end-to-end flow of what producing a hackathon event with the Yeti platform looks like.

**Before the Event**

Prior to the event, the organizers will initialize the event within the Yeti admin dashboard, providing basic information such as name and dates. Internally, they’ll also be able to provide various information about the event to the Yeti core to determine which anti-cheat systems will be utilized. The organizers will import their attendees information (including email addresses, etc) or can utilize a SSO or open-enrollment system for attendee registration. Attendees will be grouped into teams by organizers based on the event’s requirements. Organizers can also configure their milestone tracking rules, indicating how often they would like to see partial code check-ins.

Attendees who are invited will be given early access to the Yeti platform, enabling them to log onto the site. Users who are unfamiliar with Yeti and Yeti’s layout may be prompted with helpful getting started information. Users may participate in an example project to familiarize themselves with the platform, prior to the event. Attendees may fill out some personal details about themselves, including their name, photo, contact information, et cetera.

Judges and interested sponsors will typically be provided with access after the event starts. All viewers may visit onto the public-facing page for the Yeti event, though little information will be visible prior to the event.

**During the Event**

Once the event begins, organizers will have administrative level access to the entire event, including viewing all in-progress projects and work completion by each attendee. As attendees continue to complete work, reports will be visible for organizers to view, and notifications will be generated in priority order. Organizers can monitor the event’s overall progress, address issues that arise quickly, and dismiss these reports.
Attendees will now be able to start their projects, initializing repositories and organizing their project in the most sensible way for their team and desires. Code-based work will be submitted via a familiar Git interface, allowing the teams to continue using their usual development tools and workflow to check in code. Non-code work, such as design documents, demonstration videos, et cetera can be submitted via a separate release system online, or embedded within repositories at the team’s discretion. All submitted work will be sent through the Yeti pipeline regardless of format. As teams continue to work, they will be able to track their own progress via the milestone system. If an issue is discovered via the Yeti pipeline, attendees will either be directly warned or it will be presented by an organizer, depending on the severity of the issue and event configuration. Towards the end of the development period, attendees will submit their final project artifacts to close out their work and submit it for judging.

On-site sponsors may access the Yeti event at this time, being presented with highlighted teams that have high activity or otherwise may be interesting to the sponsor to visit. The sponsors can see an overview of their work, as well as physically meet the team hands-on to chat with them about their progress. Judges have a similar level of access, but their official judging review begins at the end of the event.

Viewers may continue to visit the Yeti page, having access to the work that is made public by each attendee team. Some attendee work may be publically hidden, for example to meet closed-source intellectual property requirements. For the purposes of judging the competition, sponsors, judges, and organizers always have full access to submitted work during the timeframe of the event.

**During Event Judging**

As soon as the development period of the event ends, organizers will be prompted to prepare final reports to send the projects for judging. At this time, any major Yeti violations must be processed and closed. Minor violations or violations that cannot be appropriately resolved will be left in the internal report, and visible to judges for their discretion. Once judging is complete, organizers can prepare final reports to share the results with the attendees and public community.

Attendees will be open to continue working on their projects, however continued work will be remarked differently than work completed during the development period. Depending on the rules of the event, this work may be invalidated during judging, or simply remarked as completed after the fixed period. Primarily, attendees during the judging period will be presenting their work and awaiting for the results from judging.

Judges will view the projects in whichever judging system that the organizers feel is appropriate - this can be keynote based, expo-style, or fully virtualized. Yeti doesn’t define a fixed judging
system, as it greatly varies depending on the size, speed, and style of the event. As judges view each project, they will be able to see the finalized Yeti report for each project, including commentary on the overall submissions, risk analysis, and development history. Additionally, they will be able to view all of the project-related information created by the attendees, including demonstrations, textual descriptions, and imagery.

Sponsors will be able to view full projects at this time as well, similarly to judges. Their perspective here will be more focused on the project information and attendee backgrounds, as sponsors are typically looking to more fully understand the background and resume of each attendee for recruiting purposes.

Viewers will be able to see the final submissions and closeout event information, as determined by the organizers.

**After the Event**

Once the event has officially concluded, organizers can mark the winning projects to feature on the event page. Any remaining tasks, such as archiving day-of event details, preparing final event conclusions and post mortems, etc can be completed by organizers at their leisure.

Attendees will be prompted to export their work in order to carry it on beyond the Yeti event, in most cases by mirroring the code-based work to an alternative Git server, such as GitHub.

Judges and sponsors will continue to have access to the attendee information and their contributions, for recruitment purposes. They may value looking at the event winners to help them with their recruiting decisions, or they may value the results of specific sponsor track prizes, if present at the event.

Viewers will be able to view the winning projects and their final submissions, as well as the general event information and history in perpetuity, for the lifecycle of the Yeti event after it is finished.

**Go To Market Strategy**

Yeti is intended to be developed as open-source software, available to users who want to utilize the platform for free, in a self-hosted model. However, a core aspect of what will make Yeti successful is based on the large data volume that we’ve painstakingly collected to train Yeti to identify plagarised projects. Due to this data collection having significant costs associated with it, we will offer access to this dataset, alongside a managed instance of Yeti’s software, for a fee to corporate hackathons. However, with the partnership of Pinnacle and in gratitude towards our partnered collegiate and highschool hackathon events, we will offer the platform at no cost
to our partner events, to both enhance their hackathon organization experience and improve the overall quality work in the Pinnacle ecosystem.

For the revenue-generating Yeti managed service, exact pricing will be dependent per-instance, based on the attendee count, duration, and scope of the event, and will be decided contractually on a per-customer basis.

Commentary on the open-source strategy can also be found in a concluding section (“Protecting Open Source Algorithms From Abuse”).
IMPLEMENTATION

This paper is intended to outline a proof of concept implementation of a product beginning to implement the aforementioned product requirements. This section will detail the research that was conducted into existing work, a basic understanding of the requirements for the software, and the implementation pathway that was chosen for creating the software.

Existing Work

An initial definition of Yeti is a source code plagiarism detection tool. These tools are pervasive in the software industry and academic industries. This section will note some prior work done in this area for basic source code plagiarism detection.

Beyond source code analysis, Yeti’s main goal is to be purpose-built for hackathons. This means not only scanning for copied or reused source code, but also studying additional concepts, like file metadata, project details, etc., that are untracked in industry leading competitors. From our research, there is no purpose-built hackathon anti-cheat present in the market, which is the space that Yeti is looking to occupy.

Academic Plagiarism Detection

A popular use for source code plagiarism detection is for academic purposes. The classroom environment is often high pressure, and newcomers to software development are put in situations to develop solutions to problems that are novel to themselves. In some situations, this encourages students to copy code from online, especially for well-defined topics. For example, a sorting algorithm’s code may be copied from the internet rather than replicated logically by a novice programmer. In order to thwart this behavior, many academic environments are using plagiarism detection software, to detect if submitted code is copied from another student or from the Internet.

One common example of this software is “MOSS”, a software similarity engine created by researchers at Stanford. MOSS allows users to upload single, un-linked submissions of code files in a few specific languages, and it computes plagiarism matches against its historical database (Schleimer et al., 2003). Its use is publicly available for non-commercial uses, with the commercial implementation licensed to Similix; an example of commercial plagiarism detection that will be later discussed.

Although some of the practices and design choices that MOSS relies upon are public, many of the specific details are private, for reasons that will be discussed in a later section (“Protecting Open Source Algorithms From Abuse”).
Industry Interview Plagiarism Detection

Another common place where plagiarism software is used, in a similar fashion to academic plagiarism detection, is for industry interviews. At many software companies, technical interviews are conducted that involve writing code to achieve a basic goal and demonstrate logical reasoning. Oftentimes, these goals are similar to a basic computer science homework assignment, such as “Take in a linked list as input and reverse it as the output.” Because of the nature of these assignments, it is often a risk if an interview candidate finds a solution for a question online, and then copies it. This is an especially growing problem with the growth of online interviews being conducted, especially throughout the COVID-19 pandemic.

Much of the same software that is used in academia is used for this purpose, albeit with a commercial license. Codequity is one example of an online service that particularly aims to protect unique submissions when filtering job candidates (Codequity, n.d.).

Commercial Plagiarism Detection

A final topic is commercial plagiarism detection. Oftentimes, companies will run brand-wide plagiarism detection across their publically accessible software. Outside of an academic environment, this is rarely done in an effort to ensure an engineer isn’t copying work. Instead, it’s done to preserve the corporate ownership and copyright privileges of the corporation over their own source code. For example, if an engineer incorporates a copyrighted algorithm into a company’s source code without a license, that company might be liable for copyright infringement lawsuits without their knowledge. By detecting stolen code in their own codebase, a company can filter this out to ensure they do not accidentally ship someone else’s proprietary material.

To perform these tasks, specific plagiarism algorithms are used that not only detects and matches code to its origin, but determines if that origin is licensable, or if the code is usable in a commercial context without legal issue. Mend.io is an example of a purpose-built engine with this task – it studies corporate codebases and finds their dependency code or copied code snippets, and determines if they are appropriately licensed. While many of these licenses may need to be manually validated by a legal team, the purpose is to catch the use of copied code that may be otherwise undocumented (White Source Ltd., 2023).

Technical Goals

The architecture proposed for Yeti is designed to be attainable in incremental steps. In this paper, a proof of concept is outlined and implemented utilizing the described architecture, and it is expected that the final product adds on from there.
The proof of concept’s scope is fluid: ultimately, the intent is to demonstrate the capability of the purpose-built architecture, without fully building the entire product end to end. A brief overview of the proof of concept demonstration may look like (see Appendix I):

- Demonstrate the presence of a server and client
- Execute commands via the client communicating with the server
- Create a new project via the client
- Upload files for scanning via that client
- Retrieve reports of suspicious activity via the client

Specifically, we aim to be able to upload copied code and have it appropriately analyzed by the engine with a reported value. Further robustness and testing of the software will be studied in a later section (“Results and Testing”) involving real-world trials.

Architecture Design

The overall Yeti proof of concept architecture will be divided into a few core aspects, primarily focused on the challenging portions involved in each aspect.

At its core, the software ecosystem is centered around a server (“Yeti Server”) that intakes files, project information, etc., and outputs reports, as abstracted by a web REST API. In order to demonstrate its functionality, a command line interface will also be constructed as a reference model for making calls against the Yeti Server to perform various tasks. Depending on the end use case, this command-line interface (“CLI”) client may be used directly or incorporated into another one, such as the Yeti Git frontend interface.

Fig. 2: A general diagram of the complete architecture, including some elements beyond the scope of the proof of concept.
The Yeti CLI is a simple application that replicates the behavior expected of many CLIs, using a hierarchical command structure to easily accomplish a variety of tasks. In addition to constructing REST calls, the CLI serves as reference code for what specific tasks should do, with regards to authentication, data collection, and payload structure. Serving as a reference implementation, systems that are dependent on the Yeti Server interaction may instead execute commands against the CLI directly, until their client behavior is fully developed. In this fashion, it serves as a definitive source of truth for how interactions with the server should be constructed.

A common interface to compare against is the AWS CLI created by Amazon Web Services in Python and widely available in Linux distributions. While the CLI is relatively robust and enables a large subset of behavior, ultimately it is merely a CLI shim in front of the AWS REST API. For example, AWS CLI may locally store and manage authentication sessions, as well as handle parsing local files and data when performing cloud operations. This functionality is widely regarded as standard within the CLI guidelines popularized by the open-source community (Prasad et al., 2023). The Yeti CLI operates in a very similar fashion, aiming to perform as little as necessary to fully demonstrate and qualify the server functionality.

From an architectural perspective, our CLI links against the Yeti Client library, which in turn performs queries against the server (see Appendix II).

Integrated within the server stack, we have a database abstraction to hold much of our data, including raw file uploads. While file uploads may or may not be included within a database system, they are included within our abstraction. To fulfill the abstraction, we thus rely on both a DBMS software as well as a large-file storage software. The Yeti proof of concept will rely on PostgreSQL as the primary DBMS, while utilizing PostgreSQL’s large file storage systems to store raw file uploads.
A basic proposal for the proof of concept database is thus presented in Fig. 3. Of note, a project exists and defines a single submission of work. Each project consists of a series of project_scans, which represent a snapshot of the contents of that project at a given time. A scan thus has a large amount of files, but those files are abstracted through specific instances of the file associated with the scan, in file_instance. This is because a file itself is represented by the exact hash of the file contents, and entirely identical files may appear across different project scans. As such, a specific project scan is only associated to an instance of a file, and not the file itself. When we create a new file, we also have data associated with that creation, including a table of fingerprints used to identify the file source, and a separate binary table for looking up raw contents of a file.

The exact field values in this database are fluid and will vary as proof of concept development grows and new values are introduced into the system. However, this is described to be the barebones requirements for achieving the technical aspects desired by the Yeti proof of concept. A few key details to track include:

- The file_fingerprint table: this table represents all of the small, partial hashes involved in a text file. Specifically, a file is fingerprinted by dividing up the file’s contents and creating a unique hash that represents that segment of contents. As such, two fingerprints will match between files if the corresponding segment of code matches.
- The exif_vars field: this is a generic indexed JSON field allowing for EXIF / file metadata to be imported into the database and tracked for non-text files. Since there’s a large

---

**Fig. 3:** An Entity-Relationship Diagram depicting a proof-of-concept database structure
amount of binary types, this data type provides ultimate flexibility when storing and matching two similar binaries with arbitrary metadata contents. These two concepts will be the source of primary file analysis performed by Yeti’s core pipelines, as will be later described.

In order to facilitate some of the additional operations of Yeti, like actually converting fingerprint data into cohesive matches tying two files, additional tables will likely be generated in the future. However, this diagram serves to lay out the core database elements that fuels the costly data input actions in the Yeti pipeline flow. Additional tables can also be produced by plug-in modules, as later described in the Yeti Engine section.

Yeti API

As previously mentioned, the Yeti API serves as the boundary between the client and the Yeti Engine. Its functionality is designed to be simple. It receives streams of data over a standard, REST HTTP/s standard, and outputs data similarly. The API handles some of the basic business logic, such as establishing projects and scans, answering GET requests to retrieve stored information, et cetera. However, a majority of the heavy lifting is transparently passed onto the Yeti Engine - new file uploads, or other behavior that warrants the plagiarism engine to run, is queued by the API and then further handled out-of-band from the web task. Since the management is out-of-band, clients are expected to push data to the API, and then query for a result in the future. Return out-of-band responses via webhooks are also possible, but will not be covered within the scope of the proof of concept.

This model is comparable to the Grafana data visualization software model created by Wikimedia and numerous open source contributors. In the Grafana stack, many applications exist that run in parallel, like Grafana for visualization, Prometheus for a database store, and Prometheus Exporters for data sources (Volz, 2023). All of these services run as completely separated apps, but they interconnect using standardized REST APIs between each service, even when sending large streams of data between each one.

While custom protocols are possible to better facilitate this flow of data, such as direct UNIX socket connections, ultimately the Internet has developed capabilities to allow for the small, excess overhead introduced by HTTP’s form factor on top of TCP. While particularly high-bandwidth and high-speed traffic is left to run on purpose-built protocols, such as the DB-PostgreSQL connection, lower bandwidth data such as file uploads and report downloads remain on the HTTP/s protocol for simplicity in creating joining interfaces.

The Yeti API’s primary I/O encoding will be JSON due to human readability and debuggability. With the tightly coupled client-server architecture, it’s also possible for a streamlined syntax like
Protobufs to be used instead of JSON encoding in the future, with little change required. Large data handling such as file uploads will be handled in the web-native octet stream, just the same as if you used your browser to upload a file to a website.

Yeti Git

Fig. 4: A marketing screenshot of a modified Gitea interface with Yeti details added on

A previously described “YetiGit” interface also warrants technical discussion. Although it is not within the scope of the early proof of concept deliverable, it will ultimately serve as the key interface used by the end user. At the proof of concept level, development is concentrated on the command-line based tools that establish the core design of the Yeti API. From a product standpoint, the Git interface is synonymous with the product itself. As such, the external name is truly just “Yeti”, a term that externally describes the entire system, including the front-end and back-end stacks.

The Git system is an overloaded term, in this case. Our implementation will provide the functionality of a Git server, an API controlled by the popular “git” CLI tool. Separately, it will
also provide hosting functionality for a web interface that allows web-based exploration of the Git system. This is how many hosted Git systems work, notably GitHub and GitLab (GitHub, Inc., n.d.).

Since it is not within our technical scope to create a managed Git server from scratch, we intend to pull our dependencies from Gitea, a popular open-source Git system whose MIT license permits extensibility and repurposing the Gitea source code to be a functional product (Xiao, 2023). In the scope of Yeti, in addition to providing basic brand identity and theming across the Gitea platform, we also intend to add a few of basic functions that outline the key Yeti functions: Yeti Engine hooks, event-platform controls, and project submission.

Yeti Engine hooks are the biggest driving factor for spinning up our own hosted Git system. Ultimately, on existing platforms, there is no effective way for us to manage attendee repositories with a level of control that would be satisfactory toward the inspection that Yeti requires. GitHub allows for repos to be created with “Actions”, a set of CD/CI scripts that run per-commit; however configuring these to effectively target Yeti’s API relies on user cooperation within the repos (Daigle, 2018). By running our own Git server, we can provide an interface that is familiar to users (as Gitea’s core interface feels very similar to GitHub and GitLab), while also owning the entire Git server stack, guaranteeing that we capture every commit and action performed by the attendee. This allows us to quickly and unceremoniously deal with issues of the Yeti Engine being improperly invoked by the attendee, or access permissions not being appropriately configured to allow Yeti to access the code throughout the event.

Further, Yeti needs a centralized interface to centralize Engine information for attendees and organizers. Organizers will want to view reports and configure event-level settings. Attendees will want a way to effectively know if they’re within the expectations of the event. A centralized Git interface allows us to stay close to project code while discussing other details about a project, without requiring a third party interface to be involved throughout the course of the event.

Finally, since Yeti maintains a close relationship between projects and their underlying code, it is sensible for project submissions to be managed via the YetiGit interface as well. Ultimately, attendees can prepare a presentation deliverable consisting of their source code, project description, and demo files in a single, coordinated place. It is possible, for preservation, that this data can be duplicated on a common store as well.

**Yeti Engine**

The Yeti Engine is the true heart of all of Yeti’s construction. The engine primarily contains the “Yeti Pipeline” - a step of core, fundamental actions performed on every incoming file in order
to perform plagiarism detection against it. The proof of concept development of Yeti pushes the desire for five basic pipeline steps, defined as Collecting, Unifying, Slicing, Matching, and Reporting. In order to support flexible development of Yeti, we have constructed these 5 steps into “stage types” and proposed a default sequence of actions for each type, called a “stage”. A given pipeline is able to run the default stages, as well as take in plug-in stages at each of these stage types, dictating its specific action and where in the flow it touches data. A marketing visualization can be seen in Appendix III for a summary of the Yeti pipeline.

In this section, the default stage implementations will be described in order to detail the fundamental purpose of each stage type.

Pipeline Stage: Collecting

The collecting stage is the first step of the pipeline process. Since much of the pipeline is dedicated towards source code analysis, this is the stage that is dedicated towards binary analysis and filtering. Specifically, it’s designed to catch binary files, analyze them, then archive the data immediately without proceeding to the rest of the pipeline steps. As will become evident, the remaining pipeline steps are only sensible for text / source-code based data at this time.

Specifically, the collecting stage uses “magic bytes” contained within various files in order to identify if it’s a text file or a binary file. Most binary files are encoded with a descriptor near the beginning of the file to indicate what the file is; essentially, header data (Kessler, 2002). For example, Linux ELF files begin with the literal letters “ELF”. The PDF specification dictates that a PDF file should begin with “%PDF”. If we match against a known binary type, we can specifically target it for analysis by reading document-specific metadata. If we don’t have a known binary type, we can also deduce if it’s a binary by reading the file and determining the contents of printable characters within. For example, most binary files contain arbitrary bytes or raw instructions, and a smaller percentage of printable characters, while most source code files contain only printable characters.

In this stage, we specifically are utilizing the popular and robust “exiftool” open source software to identify binary files and read their metadata. Although originally purpose-built for reading “EXIF” data embedded in photograph image standards, the tool’s use has since been expanded to read metadata from all sorts of different files (Harvey, 2023). While ultimately Yeti should be dependent on an internal source for this resource, in order to deliver on an efficient time-to-market proof of concept, we will be relying on the “exiftool” executable for collecting this data.
In this stage, metadata is then described in the file’s database entry, and the file is then discarded. Due to a lack of utility and large size, binary files are not preserved in our database besides for their metadata. In some cases, this tradeoff may not be acceptable, and large file storage should be developed to appropriately handle and store these types of files long-term.

Pipeline Stage: Unifying

The following stages now only deal with source code files. The unifying stage is designed to “unify” code that is written in multiple different languages, in order to equate matches to a common or shared theme. Specifically, this step handles language-specific tasks in order to improve the matching capabilities of later pipeline steps.

For example, a variable literal in Scala is declared as:

```scala
val myInt: Int = 3
```

wherein Python, the same may be declared as:

```python
myInteger: int = 3
```

The literal hashes of these two snippets differ since the syntax differs dramatically. A unifying engine may look at a few key details in each language, to determine a common syntax to match between the Python and Scala snippets. For example, take the thinking:

1. A variable name can easily be changed and doesn’t impact the algorithm. We should rename all of our variables to “VARIABLE1”
2. In Scala, our variable is defined as a constant. Generally, this wouldn’t affect an algorithm, but provides some safety towards the code. Since we care more about the textual similarity and less about the executed similarity, we can remove this information.
3. We use an “Int” in Scala, and an “int” in Python, but the same value may be represented as a “Number” in another language. Ultimately, we care that this is a numerical value, so we might concretely define it as a “number”. Similarly, “int64” or “float” would be just a “number”.
4. Both languages use a common “: type” syntax to define a type hint - let’s concretely use “int variable_name” instead, matching the common C-style syntax. Some type interpolation may be necessary in cases where the language does not require strong types, as could be possible in the aforementioned Python sample if typing was not included.

With these rules applied, the rewritten code becomes:

```scala
number VARIABLE1 = 3
```
number VARIABLE1 = 3

In this sample, these two lines are now identical and excellent candidates for performing a literal hash against and fingerprinting in future steps. This sample is somewhat contrived, but the unification step’s purpose is clear: use language-specific knowledge to transform a language to look familiar textually, even if not identical.

The language-specific step also has several other proposed features. For example, creating an index of all string portions (string literals, variable name literals, key/value literals, comment literals). It’s not uncommon for stolen code to keep the same comments yet shift around the literal syntax, and archiving symbols (variables and function signatures), comments, and literals here is a useful strategy for catching those easy flags. However, language-specific actions are often required to systematically denote these values: some languages will identify a comment with a double forward slash, some will use double dashes, and some will use a pound. A string literal could be denoted with a double quote, a single quote, or a sequence of grave characters.

Another proposal is to perform language-specific syntax reading – for example, instead of just modifying the language textually, we can use a purpose-built analyzer to build an execution tree of the source code, and compare this tree nodally to generate matches. For instance, two recursive sum functions may be written in slightly different syntaxes, but their functional purpose will be revealed by this analyzer and compared.

While developing a unifying engine, an important step is to avoid creating a syntax that is unified and scannable for some languages, but entirely unmatchable against others. For example, if we can only actively support 4 languages for unification, we need to ensure that the remaining possible submission programming language options remains very broad, even if the unification support isn’t as verbose or rugged. Some alternative projects, like MOSS, have fallen into the trap of explicitly offering support for a specific subset of languages, although offering a broad set of 25 common languages with support.

A proposal for offering a wide subset for basic unification is to utilize the extremely large syntax highlighting libraries popularized in programming IDEs. In 2005, popular text editor TextMate launched on macOS (then Mac OS X) with a built-in, plug-and-play “Language Grammar” system that involved creating concrete definitions for identifying different parts of a source language’s syntax to perform coloring operations (Odgaard, 2005). A massive army of language grammars, for almost every language imaginable, were quickly community developed and served as the basis for future IDEs to reuse TextMate’s bundles and incorporate them into modern editors, like Microsoft’s VS Code, many years later.

Similarly to how IDEs utilize language grammar files to perform basic token identification for unique coloring, Yeti can incorporate the same definitions to extract regions of code that
indicate certain types of tokens. For example, a TextMate grammar has a standard of declaring strings with an identifier “string.*”, wherein the asterisk is replaced by an identifier describing the type of string (a single-quoted string, a double-quoted string, etc). Conventions are maintained so that a common coloring theme can utilize the grammar to color different languages similarly - in our case, we would be identifying different segments of code similarly across multiple languages.

This approach offers us very quick, early growth in supporting a large number of first-class languages with minimal specialized development. However, as the use case grows, it may prove valuable for us to specially invest energy into focusing on a few key region-specific regions to perform additional processing (for example, JavaScript and Python are both very common languages within our domain).

Pipeline Stage: Slicing

The slicing stage is in charge of generating the actual fingerprints related to a piece of source code. It’s specifically called slicing because it performs the job of taking code, dividing it into substrings, hashing each of those substrings, then archiving the results. Slicing itself is a relatively simple task, but it leaves a few questions to be answered: slice length and hashing method.

Slice length must be determined based on the source data. It specifically describes the length of the substrings to hash – for example, are we hashing 5 characters of code at a time, or 15 characters. There’s a large variability in the choice here, and ultimately the answer depends on the input data syntax. If the syntax is very verbose and involves a lot of complicated symbols, then a larger substring will be needed to gain valuable data in each symbol. If the syntax is very compressed, then a larger slice length may cause us to hash unrelated code together, making the likelihood of finding matches decrease.

Ultimately, we must aim for the language-specific unification task to generate a common input theme here. Code written in an esoteric compressed language like Brainfuck will never slice the same as code written in an extremely verbose language, like Inform 7 (Nelson, 2006; Raiter, n.d.). We will further use empirical testing to determine an appropriate slice length based on our unification. It is important that slice length is built to be permanently fixed, as the validity of the fingerprints is entirely based on the length of the slice (i.e., a 5 character long fingerprint will never match a 6 character long fingerprint in a meaningful way).

Next, our fingerprinting method relies on a hashing function. That is to say, we will want to intake a unique string, and output a unique hash that identifies the string. There’s a few important details to note: first, we will be computing a large amount of hashes, so we want an
algorithm that is efficient and quick to not stall processing. Second, our hash ultimately needs to be smaller in size than the data we’re hashing itself. Otherwise, there’s no point in generating a hash that’s larger than just matching against the actual substring. Finally, our hashing method doesn’t have to be perfect at avoiding collisions, as ultimately matches will be generated based on a series of valid hash matches, not just a single unlucky collision. However, of course, the hashing space should be large enough to avoid errant collisions in most scenarios.

The implementation details of the hashing method depend on empirical results based on processing speed, match lookup speed, et cetera when generating hashes. However, some existing work has been done to suggest specific algorithms. For example, a rolling hash is often used to compute a single hash, then shift a data window over and produce a new hash purely by modifying the part that left the window, and the new part that entered, while not re-computing the middle (Neumann, 2015). For example, “hello” may have a hash of $1+2+3+4+5$ (15), and “ellot” may have a hash of $2+3+4+5+6$ (20). The matching portion of the two words does not need to be recomputed - instead, we subtract the “h” and add the “t”, yielding the final hash of $15 - 1 + 6 = 20$. A popular rolling hash method is the Rabin Fingerprint, which is a partial string fingerprint designed explicitly for the types of matches that we’re aiming to do here (Karp & Rabin, 1987).

Ultimately, Yeti must run a case study on different possible hashing algorithms to find one that performs reliably and consistently on the product end, while also performing quickly on the technical end. On modern systems, especially with hardware acceleration, generating hashes of short text strings is a trivial operation, however at scale it remains to be seen how efficiently we can accomplish these tasks.

**Pipeline Stage: Matching**

After fingerprints have been generated, the matching stage is responsible for aggregating the fingerprint information and comparing it against other files. Specifically, it needs to perform the database transactions to aggregate the known fingerprints for the current file, and search against all previously known fingerprints in order to find a match. Performing this task in a systematic manner is a relatively high performance task that may require a high degree of computing to scale effectively. Alternatively, an algorithm can be used to reduce the amount of data that is needed to be cross referenced at any given time. For example, if one fingerprint match is often grouped with numerous other matches around the same area, we could only match against a smaller subset of the document’s fingerprints, then rely on the full fingerprint set for performing exact comparisons against a smaller set of documents that passed the first pass filter. One of the published papers supporting the MOSS algorithm describes the “winnowing” algorithm, which is similar to this (Schleimer et al., 2003). The winnowing
algorithm defines a concrete way to reduce the number of hashes stored in a document – instead of storing every single substring, we might instead store 10% of the substrings, selecting them using a method described within the winnowing algorithm. The matching algorithm we describe is instead “double-winnowing,” by performing a winnowing procedure on the hashes we initially intake, as well as performing double-winnowing during the hash matches later performed.

For the Yeti proof of concept, the time complexity of performing a match will not be an immediate concern, due to the relatively small sample set of data, and our flexibility on timely results being produced. However, at scale, it is a certain issue that needs consideration. With a large amount of compute resources and well developed queries, it is possible that we would be able to maintain full fingerprint queries against the database. If not, however, a double-winnowing strategy can be applied as described above as a first-pass filtering strategy. This will be further discussed in the “Results” section.

For most of this section, we discuss matching fingerprints, but another innate matching step is to match and flag binary metadata that was previously collected during the collecting stage and search it against the database, similarly to fingerprint data. Although new flag factors for metadata will be added over time and based on user experience, the core concept is to highlight property similarity. For example, two binaries that share millisecond identical compilation times would likely draw attention to study the overall similarity between their two source projects more closely.

Since the matching stage is responsible for raising all flags before the report generation, it also performs event specific tasks, such as looking at the timestamps included in submitted metadata and raising an issue if it is out of the scope or rules of the origin event. For example, a PDF file with a creation date from a month before the event’s start date would raise a “match” which would later turn into a report.

Pipeline Stage: Reporting

The final stage of the pipeline is report generation – this is what collects the data from the matches and generates final reports to be presented about the project. In this stage, overall sensitivity towards matches being coincidental versus plagiarism is defined, as well as determining other behavioral changes based on the project-specific configuration.

For example, the matching stage may flag that two code files have a few overlapping hash segments. The reporting stage then must look at those overlaps, and determine if the matches warrant plagiarism, and if so, how severe it is. While these will be generally controlled on a
system-wide level, there are also opportunities for event-specific sensitivity parameters, to tune the reporting as necessary based on the rules and requirements of the operating event.

Ultimately, it’s the Yeti Engine’s responsibility to generate reports. However, the validity of those reports is still determined by the specific user needs, and the visibility is determined by the end interface that is utilized. For our proof of concept, we’ll view all generated reports and manually determine which reports are valid and which can be ignored via the CLI. In the long run, we’d likely need to generate an aggregate statistic that can be used to predict the severity of each report. An event runner is interested in only chasing credible threats to the event’s integrity, and drawing user attention for every possible match is not an effective use of user time.
RESULTS AND TESTING

In order to effectively manage testing this software, testing stages have been divided up into segments. Within the contents of this paper, the first test stage, “Functional Testing” has been completed with documented results. The remaining test stages occur after the release of this paper. As such, the plans for the stages have been outlined, with no yet recorded results. These stages follow a standard alpha, beta, release candidate, GA versioning scheme.

Functional Testing (Alpha)

In this test stage, we will outline how Yeti’s proof of concept can be demonstrated and tested for basic functionality validation. These tests will demonstrate the effectiveness of the software in archiving data, matching submissions against the archive, and appropriately raising reports.

Test Requirements

In order for this stage of testing to be considered a success, the following feature set and objectives must be met:

- **Interface**: An interface should be present to observe the results of the application as it executes.
- **Basic Handling**: The platform is expected to be able to intake files in some form and export report data as a response.
- **Submission Grouping**: Files submitted to the platform should be grouped together in a logical arrangement to group a single project from multiple projects, and a scan of a project from previous scans of the same project.
- **Textual Fingerprint Archival**: New text files (code, etc.) that are uploaded should be archived into permanent storage, and fingerprints should be saved.
- **Identical File Matching**: Files that are identical in two projects should raise a match, with the oldest file having precedence as the true “original”.
- **Partial File Matching**: Text files that contain text which is also found in previously uploaded files of different projects should raise a match.
- **Match Reporting**: Matches should be reported in a way that provides evidence for the user to further investigate and be recorded for future access.

Test Plan

To meet the demands of the alpha stage of testing, a basic test for fingerprint validity and matching will be performed, and a robustness test will be executed in order to assess the performance of the algorithms and systems involved in the Yeti engine. The basic fingerprint
validity test will serve as a baseline to begin to see if Yeti is effective at building and indexing across a database of fingerprints. Ultimately, field effectiveness in this area is dependent on a large data set, which will be later tested in the beta regression. Further, before a large dataset can be passed through the Yeti system, it is important to validate if the algorithms used are robust enough to scale for future use, or if they need to be further optimized. Some algorithms used, such as the hashing method for each fingerprint, will need to be solidified before large datasets can be implemented, since they must remain constant for the lifecycle of the system.

To facilitate the needs of the basic fingerprint test, an example algorithm is implemented in code, and then rewritten several times to generate possible alterations. Success of the system would demonstrate that the alterations are identified as stemming from the same base file. In a robust system, it would be expected that this analysis would be performative even in a database environment with a large amount of other files introducing noise.

To test the robustness of the system, two tests are proposed. The first introduces a large number of arbitrary files into the system, each with unique contents. In this test, the timing for each stage will be recorded and documented to see the rate of slowdown experienced by the system as the number of files grows. It will always be expected that more files means more possible fingerprints to search through, and thus a system slowdown will occur. However, quantifying that growth rate and which algorithms contribute the worst is an important metric. The second test is a continually growing file test. In this test, a single file will have bytes appended onto it and re-submitted over and over again. In each iteration, it is expected that the already existing segment of the file will have matching fingerprints, and new fingerprints will be generated for the new segment of the file. In this fashion, the number of reports will be far greater than in the first test, since each file uploaded would expectedly generate a report against every single previous file uploaded. This serves as a heavy stress test on the system’s ability to generate and document file fingerprint conflicts.
Test Results

Fig. 5: Two similar merge sort implementations are shown and ran through the Yeti system.

```typescript
func merge(a []int, b []int) []int {
    final := []int{}
    i := 0
    j := 0
    for i < len(a) && j < len(b) {
        if a[i] < b[j] {
            final = append(final, a[i])
            i++
        } else {
            final = append(final, b[j])
            j++
        }
    }
    for ; i < len(a); i++ {
        final = append(final, a[i])
    }
    for ; j < len(b); j++ {
        final = append(final, b[j])
    }
    return final
}
```

```typescript
func mergeSort(items []int) []int {
    if len(items) < 2 {
        return items
    }
    first := mergeSort(items[:len(items)/2])
    second := mergeSort(items[len(items)/2:])
    return merge(first, second)
}
```

Fig. 6: The corresponding analysis is shown in the Yeti CLI.

In Fig. 5 and Fig. 6 above, the aforementioned fingerprinting test was executed. First, two merge sort algorithms were written that are very similar to each other, with some code re-ordering and variables renamed. The base submission was uploaded into Yeti and then the clone was uploaded. The engine identified a report of 7 matching segments, which are the colored regions in the Fig. 6 above. In this test, there is no code unification stage, so whitespace and variable
names are left unaltered. In areas where the variable names have been altered, there is thus some expected non-detected copying. However, a large majority of the code is recognized as copied between the left and right instances. In this way, the system successfully identifies a match. As a baseline, this execution took the algorithm 160 ms in total to execute on average. This is an isolated environment with little noise - in a high-noise environment with a large amount of files, file processing time rose expectedly. See the following robustness test for more information.

The aforementioned robustness testing was performed. The first test relied on 10,000 submissions of random text to be inserted into the database, with each stage being timed.

![Moving Averages of Processing Time Per Stage](image)

**Fig. 7: Processing time for each stage when submitting 10K randomly generated files**

In Fig. 7, we can see the processing time breakdown between each stage as we upload 10K files back-to-back. Initially, it can be seen that there’s some time fluctuation on both Dactyloscopy and CodeSlicing stages - almost like a warm-up period. Then, around 2000 files, basically all stage times are near zero except Dactyloscopy which is clearly growing over time. The Dactyloscopy stage is responsible for finding aggregate matches between a set of fingerprints and the entire fingerprint database. As such, it makes sense for this to grow over time - if an average file has N fingerprints and there are X already existing files, then a new file requires requesting N*X fingerprints from the database. A major cost of this transaction is performing this fetch within an SQL request with disk caching, since it basically requires holding the entire fingerprint table in memory otherwise. While growing over time is expected for this query, it needs to be optimized to grow at a much slower rate or it will not be scalable. In another test with progressively growing files, we can see this issue blowing up even more dramatically.
In Fig. 8, “stacked” files are progressively uploaded - that is, a file that appends random data to the previous iteration of the same file. In this sense, the files grow quite quickly, generating a large total number of fingerprints. As they grow, we can see the Dactyloscopy stage blow up severely in time, requiring 8 seconds per query with a large amount of fingerprints in the database. This processing is entirely unacceptable and highlights the inability to process large sums of fingerprints efficiently.

The “Dactyloscopy” algorithm is specifically using a Jaccard similarity algorithm to detect similar files. In a SQL query, it takes the set of fingerprints from an uploaded file and compares the union and intersection of the set of fingerprints for every other file in the system. This is extremely slow to process in SQL, especially as the number of fingerprints exceeds bandwidth for an in-memory transaction. In order to combat this, a first-pass algorithm is likely necessary to thin out a large amount of fingerprints to a concrete set of possible matches to look at. A locality-sensitive hashing algorithm is appropriate for this purpose, in order to summarize the Jaccard similarity without actually transacting the entire set for each file. An algorithm like MinHash or SimHash could effectively do this, by picking the smallest fingerprints in each set when computing similarity (MinHash) or summarizing the similar fingerprints by value in a single hash (SimHash). Either method could reduce the computational complexity drastically and greatly speed up runtime. Both algorithms have pros and cons, however, and should be studied practically to determine an ideal path forward. This slowdown must be rectified within the system before progression can be made to the Beta stage.
Pre-Production Testing (Beta)

In this test stage, the feature set from the proof of concept is expanded to prepare the software for production use. This includes plans to expand robustness and test that growth. In order to make the Yeti software effective for public use, this stage will involve “training” the system on a wider set of data.

Test Requirements

In order for this stage of testing to be considered a success, the following feature set and objectives must be met:

- **Trained Dataset**: A large set of existing projects should be uploaded into the system, providing a more real-world set to compare against.
- **Binary Metadata Archival**: New binary files (videos, etc.) that are uploaded should have their metadata recorded and permanently stored.
- **Partial Metadata Matching**: Binary files that contain metadata values which are also found in previously uploaded files of different projects should raise a match.
- **User Friendly Interface**: An end user experience must be crafted at this stage to facilitate field trial testing, including necessary authentication and administrative systems.
- **Robustness at Scale**: Robustness at scale must be tested and validated, including parallel operation and handling for real-world traffic.

Test Plan

This stage exposes the Yeti system to the first real-world tests. A large dataset can be aggregated from existing hackathon projects and reference sources. One proposed dataset is from Devpost, an aggregate website that tracks most hackathons worldwide and their project submissions. Another popular dataset is from popular websites where code is copied from, such as StackOverflow or GeeksForGeeks. Finally, popular open source libraries that are pulled into projects such as jQuery or TensorFlow should be studied as well. While these libraries are often not fraudulent to be included in a project submission, they should be pulled into Yeti and marked as a known library so that Yeti can discard matches between projects that utilize the same libraries.

Binary data processing should also be added, in order to appropriately perform Yeti analysis on non-code files. By studying metadata from different files, we can uncover hidden details that may otherwise go unnoticed by both staff and fraudulent users. Uniquely identifying metadata, like a video’s encoding timestamp or a binary’s embedded strings should be archived and cross-matched across Yeti. Other fraudulent metadata, such as a creation date before a given event deadline, should be archived and flagged as a report on an event-by-event basis. This
introduces configuration of the Yeti engine to generate both global reports and event-based reporting. Since not all hackathons share the same event characteristics and rules, event-based reporting is required to understand if a submission is fraudulent under a given set of rules.

**Field Trial Testing (Release Candidate)**

This test stage involves a field trial environment, in which the software is tested with real users at Pinnacle’s September 2023 event. At this point in time, with real users, the system is expected to be fairly robust to use. Additionally, it will be expected to perform appropriately to minimize the risk of fraudulent activity in a real world environment. During the trial, the behavior of the system will be closely studied and second-checked, in preparation for a truly automatic system for eventual release.

**Test Requirements**

In order for this stage of testing to be considered a success, the following feature set and objectives must be met:

- **Ease of Analysis**: Staff must be able to analyze the results produced by Yeti during the Pinnacle 2023 event in real-time.
- **Overall Efficacy**: The effectiveness of the software must be quantified based on the detection behavior as opposed to manual analysis performed by staff - both in terms of false positives and false negatives.
- **Overall Robustness**: Time analysis on performance should be studied during the duration of the event resulting in no spikes or noticeable delays inhibiting the user experience.
- **End User Experience**: Satisfaction surveys should be conducted as a basic user study data point with no clear issues noted.

**Test Plan**

The test plan for this stage is largely driven off of the natural reactions created during the field trial tests. Analysis includes the validation of the platform as confirmed by staff, basic satisfaction from all stakeholders, and postmortem analysis on any arising platform issues.

To validate the platform from a staff perspective, real-time testing should be done via Yeti and also manually. Any noted discrepancies, as in issues caught by Yeti or by staff but not both, should be documented and triaged at the end of the event. Two separate teams should be trialed - a team empowered by Yeti and a team empowered by manual research only. These two teams should compare their experiences after the event, both in terms of time commitment and overall coverage, to assess overall advantages of the Yeti platform.
As a part of a user-driven development flow, exit (and entrance if possible) surveys should be collected from all stakeholders - staff, attendees, and sponsors - to determine overall satisfaction, discover unrealized experience issues, and judge the success of the system end to end. The questions and data from this survey should be driven based on the overall key features demanded on the outset of product development for Yeti and should comprehensively identify successful coverage on the desired features, as well as identify new angles and approaches that were not originally realized during the original product proposal.

**Production Testing (GA)**

This final test stage involves final testing of the Yeti software before its initial general availability release. This stage is significantly impacted by the results of the prior stages. However, barring setbacks discovered in earlier stages, the primary goal of this stage is to build robustness, platform security, and comprehensive documentation for the software. Initial releases of the software will be expected by consumers to excel - as such, testing will be performed to ensure the success of the software in its premier.

**Test Requirements**

In order for this stage of testing to be considered a success, the following feature set and objectives must be met:

- **Security Audit**: A security audit should be performed and passed before releasing access to the system downstream to other hackathons directly.
- **Open Source Audit**: An open source audit, reviewing vulnerabilities and licensing concerns should be performed and passed before releasing the GA source bundle.
- **End User Documentation**: Documentation and other end-user requirements should be considered at this stage to make an initial public release.
- **Robustness**: The application must be tested on a wider range of environments and scenarios in order to simulate real-world situations that it may be used on - different hardware, architecture, etc capabilities.

**Test Plan**

The GA stage of development does not have a rigid test plan, as it is largely guided by the outcomes of previous stages. If a key new feature is realized in a previous stage, then this will serve as a testing grounds for proofing out that feature if it should be included in the initial GA release, or slating the feature for a future version if relevant.
System audits should be performed, both by Pinnacle’s internal auditing team and an external security team if budget allows. These tests are critical to reducing overall liability on the Yeti group for a public release of the software, but also serve as key tests to validating the overall robustness of the system at scale. Many issues may be uncovered, such as denial-of-service attacks that may occur when processing certain files. In this fashion, these issues should be resolved and mitigated before a release is made to a wider audience.

If it is warranted, a wider internal release may be performed before a public release. With Pinnacle’s large network of partnered hackathon events, an internal release can be facilitated by sharing access to the Yeti platform at more events to gain a larger set of data and grow the user audience slowly. Whether or not this stage is necessary, however, is largely driven by the outcomes of prior testing stages.
COMMENTARY AND CONCLUSIONS

Plagiarism Deterrence and Plagiarism Filtering

It’s important to establish a difference between the acts of making plagiarism challenging as an act, and actually detecting it. We identify the former as a “deterrence” factor, while the latter is a “filtering” factor. Both actions will prevent plagiarism downstream, however they appear very differently.

To make plagiarism challenging means to establish a context where potentially nefarious users are discouraged from participating in nefarious activity, because of high barriers to make it feasible. For example, if a well-defined coding problem is assigned to a student, it is initially easy for them to look up a solution online and provide it back verbatim. The barriers are low, and plagiarism risk is high. In the same situation, if the student is then expected to recite each line one-by-one, explaining its purpose and contribution to the overall algorithm, suddenly simply plagiarizing an algorithm becomes much more challenging, since the student must actually understand the details of stolen code, and may have just written it themselves in many cases.

In many settings, detecting plagiarism plays a role because increasing difficulty often comes with high resource constraints. In the prior example, a grader is now required to grade both the code as well as the line-by-line justifications in order to appropriately assess the work, making the grading process take longer. Instead, it is common to resort to detection systems that serve as both an inherent deterrent and downstream filter for stolen work.

In its scope, Yeti is unique because it functions as both a strong deterrent and a detection method. In their current construction, most hackathons have no way to perform continuous monitoring of attendee projects. A work period begins with the start of the event, and ends with the final project submission. However, a Yeti-enhanced event is able to perform continuous monitoring of attendee work via Git submissions, and mandate those Git submissions on a regular basis. This is not a hard concept for many hackathons to implement, however there is no definitive system in place to validate that Git submissions are occurring at regular intervals and with a reasonably sized delta per commit. Yeti’s Git intake system automatically facilitates this process, and can easily be configured to expect regular commits as a reporting criterion.

As an example, Pinnacle deployed a manual Git validation process at our 2021 flagship event. We tracked 50+ different teams, with 60+ different repos, mandating commits at 2-6 hour intervals, scheduling them reasonably based on sleeping and meal periods. This made us one of the only events to even require that participants open repositories and share access to staff at the start of the event. We had a team of engineers who scanned through the Git commits at the end of each interval, and personally spoke to each team that was out of compliance with the
schedule. This helped us establish a clear dialogue with the teams about the quality of work we were expecting as a prestigious event, as well as understand our attendee’s submissions in a much deeper way. At the end of the event, we were able to eliminate several high-ranking candidates. These were projects that were potentially award-winning, but further code review led the staff and judging team to have little evidence to be confident that the projects were fully developed during the confines of the event. However, based on plagiarism incidents that we’ve observed in the past events, we believe that active rules regarding code check-ins not only helped us identify stolen code, but reduced the incidents of teams electing to steal code in the first place – serving as an effective deterrent and filtering method.

This example reveals the effectiveness of performing simple code check-ins, a system that Yeti encourages and facilitates. However, manually performing these tasks is time consuming, error prone, and labor intensive. Pinnacle was fortunate enough to work with a relatively small set of teams and have a large amount of on-site engineers to study project code throughout the entire event. Events that are larger or have fewer free resources could drastically struggle to accomplish this task effectively. A key component to Yeti’s mission is to make this level of depth achievable at all events.

**Protecting Open Source Algorithms from Abuse**

One of the persistent threads in the open source world is how an open source design exposes a system to abuse. For example, if a server is running an open source web server, it is often argued that it is easier to view the source code of the web server, find a vulnerability, and exploit that on the actual server. Anti-cheat software widely used in video games, et cetera suffer from a similar issue – the software is usually quite invasive, but information cannot be revealed about it in fear that hackers would find a way to subvert the system. Plagiarism detection software is, understandably, similar. If the underlying structure of a plagiarism detection software is known, it is possible to find ways to bypass the system. Especially so if you’re able to run the software locally and self-test your code for plagiarism issues before submission.

This is a major factor why even academic research in this area, on projects like MOSS, are relatively secretive. Exposing the underlying algorithms are detrimental towards the system as a whole. Regardless, we set out to make Yeti an open source project in its entirety. We’ve determined this to be a low risk factor due to the hackathon nature and the decisions we made when pursuing an open source model.

The hackathon nature implies that a project must be started and completed within a specific time frame. There’s various systems on Yeti out of the box to prevent exploiting this time frame,
such as validating time stamps, preventing code reuse, and requiring code check-ins. Ultimately, the restricted time frame and environment found in a hackathon makes it challenging to fraudulently produce code that meets the Yeti standard and would not be immediately flagged, even knowing the specific checks that the algorithm is going to look for. Sequencing timely check-ins, addressing raised reports, and more are quite challenging to perform for a majority of cheaters who are looking for a low-effort, high-reward output.

Further, we’ve elected to make the open-source portion of Yeti only a partial system. While Yeti’s core is visible, the extremely large datasets we use to initiate our version of Yeti is a secret. Additionally, we have various commercial plugins that we are able to deploy within our internal Yeti servers that would remain closed-source. While the open source version of Yeti is completely usable, the setup required for it to be effective exceeds the range of a likely cheater. Overall, this allows us to create an open-source platform while minimizing the risk towards weakening our engine’s behavior.

**Future Work**

As we’ve continued the development of Yeti, multiple avenues of exploration have been opened. With immediate urgency, the development of Yeti through the aforementioned testing phases is critical to the release of the platform. However, in addition to these features, discussions regarding novel algorithms, tackling AI, and leveraging AI have proven to be relevant areas to explore as well.

Novel algorithms are seen to be critical to improving the performance and characteristics of the Yeti platform end-to-end. A majority of our work thus far has been accomplished utilizing “off the shelf” standardized algorithms. Ultimately, as can be seen in the performance results, these algorithms are not necessarily efficient or well-built for our test case. Consultation with more industry professionals and researchers is necessary to help shape and modify these algorithms to be better suited for Yeti’s use case. It is proposed that, using the plug-in system of the Yeti Pipeline, different algorithms can be trialed in direct opposition to each other. Exploring this system and the large dataset of projects, we hope to uncover new approaches to efficiently process data by creating a testing and trial framework. Much like the testing executed in this paper, a trial framework would enable us to instrument different algorithms head-to-head and judge them in terms of execution time, memory cost, disk usage, and match detection.

The use of AI in programming has exploded over the course of 2022 and 2023, with the release of popular tools like ChatGPT and GitHub Copilot exploding the use of AI-generated code. Upstream from Yeti, Pinnacle has not yet taken an official stance on the use of AI-generated code. However, the change does pose a significant challenge for plagiarism systems. Research
has indicated that AI-generated text is not reliably detectable at this time, and the same is likely true of AI-generated source code (Sadasivan et al., 2023). As this area continues to grow, it is likely required that Yeti explores detection and mitigation of AI generated code. In part, some of the plagiarism deterrence factors previously mentioned may mitigate this situation.

Additionally, AI research is also a tool that can be leveraged in favor of Yeti. It’s possible that trained extensions on existing models like GPT-4 could be utilized to better understand submitted code and apply non-deterministic AI analysis directly to user submitted code. Training an AI based on submitted code and asking the question, “Is this code similar to previous work” can yield a more holistic analysis of a project and codebase than what is currently possible with the Yeti system. This is still a growing area of research that would likely warrant further investigation in the future.

Concluding Thoughts

A proof of concept of Yeti has been developed and trialed as a system to analyze and fingerprint code submissions in hackathon competitions. Although some issues have arisen with the practicality of the current system at scale, the work and research involved serves as a strong foundation for the continued growth of the system as a commercial product. With upcoming opportunities for real-world implementation of the Yeti system, continued growth and development is expected.

In a rapidly evolving hackathon environment, with the impact of the COVID-19 Pandemic and release of AI-driven software development, the need for anti-cheat and plagiarism detection software at hackathon events is mandatory for keeping the vibrant, competitive community alive. It should be expected that online events combined with easily accessible computer-generated submissions can, and will, detrimentally impact creative ideation and project production at events. A software package like Yeti is a small step towards mitigating this transition, but is absolutely critical in the pathway to continue creating fair, engaging, and responsible competitions in the future.
REFERENCES


As a proof of concept, the above sequence diagram is proposed in which an actor performs basic Yeti operations utilizing the Yeti CLI to initialize a project, a scan of that project, track
different files, and close that scan. Ultimately the user will then be able to pull the reports out of the Yeti system in order to fulfill the expected abilities of the Yeti plagiarism detection system. While the proof of concept details operating this system entirely manually, the end system will be fully automated via various import and export automations. For example, importing data automatically from the proposed YetiGit system and exporting reports straight back onto the YetiGit visual UI.
The overall software stack is relatively simple – a shared repository of interfaces and models exist that represent the expected input/output payloads produced and received by the client and server. These are marshaled into a JSON encoding. On top of the interfaces, a client and server framework emerge. The client framework is a dependency relied on by the CLI and YetiGit application, both as end-user applications. The Yeti Server is the primary framework that connects the API and Engine together in order to make a cohesive server-side application. In specific terms, these shared structures are constructed as a shared library written in Go and linked at compile time during the production of all executable binaries.
Appendix III: The Yeti Pipeline (Marketing Imagery)

The “Yeti Pipeline” is pictured in our promotional and marketing imagery as a physical tube with different operations transforming input samples. It includes the collection, unification, slicing, matching, and reporting algorithm that is outlined in this paper.